MIT/LCS/TR-256

SEMIAUTOMATIC TRANSLATION
OF
COBOL INTO HIBOL

Gregory Gerard Faust

This blank page was inserted to preserve pagination.

Semiautomatic Translation
of
COBOL into HIBOL

by

Gregory Gerard Faust

February 1981

This research was supported, in part, by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-75-C-0643 and N00014-80-C-0505, and in part by National Science Foundation grant MCS-7912179.

The views and conclusions contained in this paper are those of the author, and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of Defense, the National Science Foundation, or the United States Government.

Semiautomatic Translation of COBOL into HIBOL

by

Gregory Gerard Faust

Submitted to the Department of Electrical Engineering and
Computer Science on January 21, 1981 in partial fulfillment
of the requirements for the Degree of Master of Science in
Computer Science

ABSTRACT

A severe software crisis is currently being experienced by the data processing community due to intolerable maintenance costs. A system is introduced to reduce those costs by the translation of existing COBOL software into HIBOL; a very high level language that is significantly easier to maintain. HIBOL, uses a single type of data object, called a "flow", which is an indexed stream of data values. Computation is expressed as operations acting on flows.

The translation process relies on a method for program abstraction developed by Richard Waters which expresses programs as a hierarchical structure, called an analyzed plan, in which control and data flow is made explicit. In this formalism, loops are expressed as a composition of stream operators acting on stream data flow.

This paper discusses in detail how an analyzed plan for a COBOL program can be translated into a HIBOL program. It is currently possible to translate into HIBOL analyzed plans for a relatively small (but well defined) subset of COBOL programs. Suggestions are made as to how that subset could be expanded through further research.

Thesis Supervisor: Dr. William A. Martin
Title: Associate Professor of Electrical Engineering and Computer Science
and Associate Professor of Management

Thesis Supervisor: Dr. Richard C. Waters
Title: Research Scientist, Artificial Intelligence Laboratory

Keywords: program translation, program understanding, COBOL, HIBOL, automatic programming

Acknowledgments

I would like to thank the following people without whose help this thesis would not have become a reality.

- Bill Martin for the seminal idea that launched the entire project

- Dick Waters for the help he gave me in writing programs which interact with his plan formalism, and for his constant guidance throughout this project

- Glenn Burke for the time he spent writing the COBOL parser, and helping me eliminate bugs in my programs which no else could fathom

- Ramesh Patil and Harold Goldberger for their assistance in devising algorithms to perform various tasks

- Dick Waters and Bill Martin, my thesis advisors, and Chuck Rich for their constructive criticism of several previous drafts of this paper

- Howard Sherman, Brij Masand, Glenn Burke, Ramesh Patil, Lowell Hawkinson, Harold Goldberger, Irwin Asbell, Bill Long, Bill Swartout, and Ken Church for their lively discussions, both over lunch and when my mind refused to apply itself to the problem at hand

- Ann Sexton for her companionship and understanding throughout the past year

- and especially, my parents, Alice and Vincent Faust, and the rest of my family for their constant moral support throughout my academic career

CONTENTS

FIGURES

1. Introduction

In the last ten years, there have been many efforts to simplify the task of producing large error-free software systems. Although no one would argue with the merits of such efforts, they alone are not sufficient to relieve the current software crisis that is being experienced by the data processing community. In addition to aids in the production of new software systems, aids in the maintenance of existing software are needed. This thesis is a step toward such an aid.

The system, SATCH, is designed to perform the SemiAutomatic Translation of COBOL into HIBOL. HIBOL is a very high level specification language in which data processing applications are not programmed procedurally, but simply described as a group of stereotyped operations acting on streams [3,18,27]. Since the HIBOL representation explicitly embodies the functional specifications of the application system, it is relatively easy to understand and maintain. The HIBOL can subsequently be translated back into COBOL [27]. The intent is that the COBOL produced by the system will be considerably more structured, and potentially more efficient, than the input COBOL program. More importantly, the HIBOL specification of the program can be retained so that future changes to the functional specifications of the program can be implemented by modifying the HIBOL program and automatically regenerating the corresponding COBOL program.

1.1 Motivation

In order to see the long term potential of a system such as SATCH, the following scenario can be envisioned. The manager of a data processing facility recognizes that one of his systems has reached the point where the code is so convoluted that it can no longer be maintained in a reasonable fashion. He would like to have the system rewritten from scratch, but he realizes the tremendous cost involved. In addition, he simply does not have the personnel to place on such a project. Without SATCH, he is doomed to live with the current system despite its shortcomings.

However, if a production version of the SATCH system did exist, he would have another alternative. He can input the COBOL programs into SATCH one at a time. For each program so processed, he gets an output of another COBOL program that is easier to understand and is probably more efficient. More importantly, he receives a HIBOL program which embodies the functional specifications for the application. (For those COBOL programs that embody computations that cannot easily be expressed in HIBOL, the original COBOL program is retained). The HIBOL program can also be utilized as documentation for the system, and can therefore reduce the need for the time-consuming production of bulky documents for the system written in some less concise form.

Future modification to the functional specifications can then be implemented as direct alterations of the HIBOL code; the need to maintain the system via modifications to COBOL programs is (largely) eliminated. The updated HIBOL program can then be used to automatically produce the newly desired COBOL program. Note that this process also updates the documentation for the system with no additional effort.

Although the process of initially converting from the existing COBOL programs to the HIBOL programs would be expensive and somewhat difficult, it would not be nearly as bad as a total system rewrite in COBOL. In either case, it is a one time expense. The benefit of the conversion to HIBOL is that the incremental cost of system maintenance is greatly reduced.

Admittedly, the above scenario will not be actualized in the immediate future. However, the technology needed to produce such a system should be available within the foreseeable future, as indicated by the level of the current technology discussed in this thesis. The component of the system introduced in this paper represents an attempt to overcome the only evident theoretical barrier. Now that this component has been shown feasible, it should be possible to resolve the remaining difficulties by further research and a lot of hard work in the form of some excellent engineering. The obvious merits of the production of such a system should make the effort worthwhile.

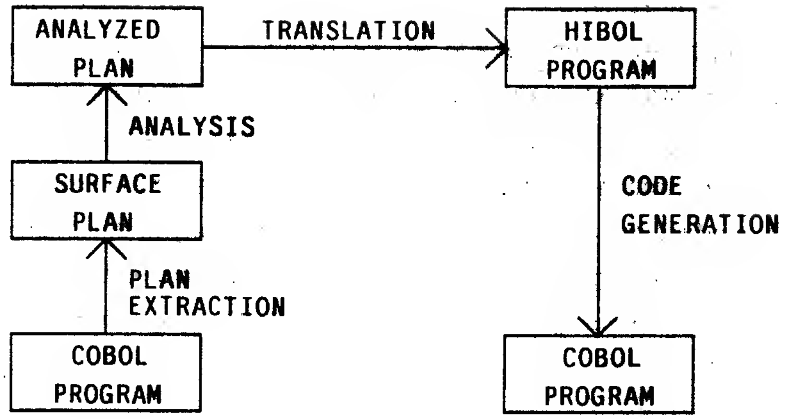
1.2 System Overview

Figure 1 is a schematic representation of the entire system. First, a surface plan is extracted from the raw COBOL code. The surface plan is then analyzed in terms of Plan Building Methods (PBMs). The analyzed plan is then translated into a valid HIBOL program. From this HIBOL program, a new COBOL program can be produced. The process that translates an analyzed plan into a HIBOL program is the novel component of the system.

The first process extracts a surface plan from the raw COBOL code. A surface plan contains all the information contained in the original code, but in a language independent form. It is a direct abstraction of the control and data flow in the original program. Enough information is explicit in the surface plan that it is theoretically possible to execute it. The original notion of a plan was developed by Rich and Shrobe [25]. The detailed structure of a surface plan was developed by Waters as part of his PhD research [31,32]. Burke and Waters have written a program that produces surface plans for COBOL programs.

The real interest in the surface plan representation of the COBOL program lies in the fact that it can be automatically analyzed further in terms of PBMs. The PBMs, the type and form of which were developed by Waters [31,32], are a small set of well defined control and data flow structures into

Fig. 1. SATCH Overview



which programs can be analyzed. He has implemented a program that produces an analyzed plan from a surface plan.

The next component of the system embodies the current research. It is responsible for the translation of the analyzed plan for the COBOL program into a HIBOL program that performs the same data processing function. It is intended to produce HIBOL code that faithfully embodies the original functional specification implicitly contained in the COBOL program.

Once the HIBOL is produced, it is used as input to a code generator. The target language can be any conventional high level language such as COBOL or PL/1. Currently, PL/1 can be produced from HIBOL by the use of an automatic programming system called PROTOSYSTEM I [27]. There are some problems with the unconstrained use of PROTOSYSTEM I to produce PL/1 from HIBOL. Within the current scenario, however, the system can be constrained to avoid these difficulties. It would be relatively straightforward to reimplement the portion of the PROTOSYSTEM I which produces the target language syntax so that COBOL could be produced instead of PL/1.

It must be stressed that this thesis should be viewed as a feasibility study. The major thrust of this thesis is to show that it is possible to produce HIBOL from COBOL with very little human intervention using technology that is either currently available or which should become available in the foreseeable future. It is not the intention of this thesis to present a final solution to the problem of COBOL to HIBOL translation.

1.3 Related Work

There have been three general approaches to the elimination of the software maintenance crisis. First, many attempts have been made to reduce the maintenance burden through the creation of more structured and constrained programming languages [15,35] that are intended to facilitate the writing of more correct programs. A second approach has been to design languages in which the program is written in a form that resembles functional specifications for the program, and then have a system automatically produce the actual code for the program. Many of these languages fall into the category of the so called "very high level" languages [7,8,27]. A third approach has been to design interactive systems in which the programmer and the system assist one another in the design of a program [4,6,16,25].

The first approach, constrained programming languages, has the advantage that once a program is written that is accepted by the compiler for the language, it has a higher probability of executing correctly. This reduces the need for maintenance aimed at assuring that the program operates according to the functional specifications. It does nothing to reduce the maintenance required when the functional specifications are changed. The second approach, very high level languages, has the advantage that the resultant programs are easier to maintain when the functional specifications change. This is a result of the fact that the programs represent the functional specifications in a more straightforward and therefore more perspicuous manner. The last approach, interactive systems, reduces both types of maintenance because the interactive system used to produce the software can be employed for its maintenance as well.

Unfortunately, though all of these approaches can be used to reduce the cost of maintenance of software systems that are implemented using them, they cannot be used to reduce the maintenance of preexisting software systems. The system described in this document, SATCH, is aimed at the reduction of the cost of maintaining existing software.

The reason that these approaches cannot be used to reduce the maintenance of existing software is that they attempt to automatically translate increasingly high level program descriptions into some lower level description. They do not attempt to translate from lower level languages to higher level ones. To my knowledge, the work of Rich, Waters, and the other members of the Programmer Apprentice Group at MIT [25,31,32] is the only effort that has been made to date to automatically produce any type of an abstraction from an existing program. It is this work that is the theoretical foundation of the SATCH system.

Some work has been done at the University of Texas at Austin by John Hartman [9] in an attempt to provide a methodology for restructuring COBOL programs into abstract data modules. Such a methodology could be applied by programmers to restructure existing COBOL programs prior to performing maintenance on them thereby reducing maintenance costs for those cases in which the methodology is applicable. The goals of the work by Hartman differ from the goals of this thesis in that Hartman's methodology is designed to be applied by a person, while we wish to abstract from an existing program via a machine. Perhaps, through further research, his methodology could be made precise enough to be automated. In any case, a HIBOL program is easier to maintain than a restructured COBOL program for the same computation.

Within the data processing community, several systems have been designed to produce COBOL from some "higher level" language based on the notion of stereotyped operations in COBOL [5,34]. None of them, however, are in the form of specification languages; rather, they are essentially macro packages or structured preprocessors. One of these, MetaCOBOL [2], can be used to translate from a COBOL program written to be executed on one vendor's machine into a COBOL program that can be run on another vendor's machine. This is merely a syntactic change, however, and does not involve either abstraction or non-trivial control or data flow alterations.

Another project in language to language translation was recently completed by Kent Pitman at MIT [23]. He wrote a program to translate FORTRAN programs into LISP. The translation is done in two steps. First, the FORTRAN is translated into a LISP form in which DO loops and other standard FORTRAN constructs are expressed as LISP macros. In the second phase, the macros are expanded into an interpretable and/or compilable form. The two step process has the advantage that the form containing the macros is somewhat maintainable, while the expanded form is much less so. Still, in Pitman's project, maintainability (and therefore readability) of the resultant code was only a secondary goal, while the maintainability of the HIBOL produced from a COBOL program was a major goal of the research described in this thesis. A more important distinction is that the FORTRAN to LISP translation is done almost entirely on a syntactic basis, while the COBOL to HIBOL translation is not.

1.4 Example Programs and Their Translations

This section presents four COBOL programs and the corresponding HIBOL programs generated by the current implementation of the SATCH system. Two of these examples, DBINIT and LOC-LIST, are programs taken from running software systems currently in use in the data processing community. Although the reader is not expected to understand the programs at this point, they are included here to give the reader a feel for the task at hand. In particular, note the large compression that takes place, especially in the translation of the PROCEDURE DIVISION of a COBOL program into the COMPUTATION DIVISION of the corresponding HIBOL program. These examples will be referred to throughout the remainder of the document. The reader is invited to turn back to these listings whenever it seems appropriate to do so.

Fig. 2. COBOL Program for PAYROLL

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT HOURLY-WAGE-IN ASSIGN TO DA-2301-S-HWI.
        SELECT GROSS-PAY-OUT ASSIGN TO DA-2301-S-GPO.
        SELECT EMPLOYEE-COUNT-OUT ASSIGN TO DA-2301-S-ECO.
        SELECT TOTAL-GROSS-PAY-OUT ASSIGN TO DA-2301-S-TGPO.

DATA DIVISION.
FILE SECTION.
FD  hourly-wage-in
    LABEL RECORD IS OMITTED
    DATA RECORD IS hourly-wage-rec.
01  hourly-wage-rec.
    02  employee-number                        PICTURE IS 9(9).
    02  hourly-wage                            PICTURE IS 999V99.
FD  gross-pay-out
    LABEL RECORD IS OMITTED
    DATA RECORD IS gross-pay-rec.
01  gross-pay-rec.
    02  employee-number                        PICTURE IS 9(9).
    02  gross-pay                              PICTURE IS 999V99.
FD  employee-count-out
    LABEL RECORD IS OMITTED
    DATA RECORD IS employee-count-rec.
01  employee-count-rec.
    02  employee-count                         PICTURE IS 9(6).
FD  total-gross-pay-out
    LABEL RECORD IS OMITTED
    DATA RECORD IS total-gross-pay-rec.
01  total-gross-pay-rec.
    02  total-gross-pay                        PICTURE IS 9(7)V99.

PROCEDURE DIVISION.
initialization SECTION.
        MOVE ZERO TO total-gross-pay.
        MOVE ZERO TO employee-count.
        OPEN INPUT hourly-wage-in.
        OPEN OUTPUT gross-pay-out.
mainline SECTION.
        READ hourly-wage-in AT END GO TO end-of-job.
        MOVE employee-number OF hourly-wage-rec
          TO employee-number OF gross-pay-rec.
        MULTIPLY hourly-wage BY 40 GIVING gross-pay.
        ADD 1 TO employee-count.
        ADD gross-pay TO total-gross-pay.
        WRITE gross-pay-rec.
        GO TO mainline.
end-of-job SECTION.
        CLOSE hourly-wage-in.
        CLOSE gross-pay-out.
        OPEN OUTPUT employee-count-out.
        WRITE employee-count-rec.
        CLOSE employee-count-out.
        OPEN OUTPUT total-gross-pay-out.
        WRITE total-gross-pay-rec.
        CLOSE total-gross-pay-out.
        STOP RUN.
```

Fig. 3. HIBOL Program for PAYROLL

```
DATA DIVISION

KEY SECTION
          KEY EMPLOYEE-NUMBER
               FIELD TYPE IS NUMBER
               FIELD LENGTH IS 9

INPUT SECTION
          FILE HOURLY-WAGE
               KEY IS EMPLOYEE-NUMBER

OUTPUT SECTION
          FILE GROSS-PAY
               KEY IS EMPLOYEE-NUMBER
          FILE EMPLOYEE-COUNT
          FILE TOTAL-GROSS-PAY

COMPUTATION DIVISION

TOTAL-GROSS-PAY IS (SUM OF (HOURLY-WAGE * 40.))

EMPLOYEE-COUNT IS (COUNT OF HOURLY-WAGE)

GROSS-PAY IS (HOURLY-WAGE * 40.)
```

Fig. 4. COBOL Program for PAYROLL2

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT HOURLY-WAGE-IN ASSIGN TO DA-2301-S-HWI.
        SELECT HOURS-WORKED-IN ASSIGN TO DA-2301-S-WH.
        SELECT GROSS-PAY-OUT ASSIGN TO DA-2301-S-GPO.
        SELECT TOTAL-GROSS-PAY-OUT ASSIGN TO DA-2301-S-TGPO.

DATA DIVISION.
FILE SECTION.
FD  hourly-wage-in
    LABEL RECORD IS OMITTED
    DATA RECORD IS hourly-wage-rec.
01  hourly-wage-rec.
    02  employee-number                        PICTURE IS 9(9).
    02  hourly-wage                            PICTURE IS 999V99.
FD  hours-worked-in
    LABEL RECORD IS OMITTED
    DATA RECORD IS hours-worked-rec.
01  hours-worked-rec.
    02  employee-number                        PICTURE IS 9(9).
    02  hours-worked                           PICTURE IS 999.
FD  total-gross-pay-out
    LABEL RECORD IS OMITTED
    DATA RECORO IS total-gross-pay-rec.
01  total-gross-pay-rec.
    02  total-gross-pay                        PICTURE IS 9(7)V99.
FD  gross-pay-out
    LABEL RECORD IS OMITTED
    DATA RECORD IS gross-pay-rec.
01  gross-pay-rec.
    02  employee-number                        PICTURE IS 9(9).
    02  gross-pay                              PICTURE IS 999V99.
WORKING-STORAGE SECTION.
77  end-of-hours-ind PICTURE 9 VALUE ZERO.
    88  end-of-hours VALUE 1.
77  end-of-wage-ind PICTURE 9 VALUE ZERO.
    88  end-of-wage VALUE 1.
77  compare-ind PICTURE 9.
    88  wage-eq-hours VALUE 1.
    88  wage-lt-hours VALUE 2.
    88  wage-gt-hours VALUE 3.

PROCEDURE DIVISION.
initialization SECTION.
        MOVE ZERO TO total-gross-pay.
        OPEN INPUT  hours-worked-in
                    hourly-wage-in
             OUTPUT gross-pay-out.
        PERFORM read-wage.
        PERFORM read-hours.
mainline SECTION.
        IF end-of-wage OR end-of-hours THEN GO TO end-of-job.
        PERFORM compare.
        IF wage-eq-hours THEN PERFORM wage-eq-hours-proc.
        IF wage-lt-hours THEN PERFORM wage-lt-hours-proc.
        IF wage-gt-hours THEN PERFORM wage-gt-hours-proc.
        GO TO mainline.
read-wage.
        READ hourly-wage-in AT END MOVE 1 TO end-of-wage-ind.
read-hours.
        READ hours-worked-in AT END MOVE 1 TO end-of-hours-ind.
```

Fig. 4. COBOL Program for PAYROLL2 (CONTINUED)

```
compare.
        IF employee-number OF hourly-wage-rec
         > employee-number OF hours-worked-rec
                THEN MOVE 3 TO compare-ind
        ELSE IF employee-number OF hourly-wage-rec
         < employee-number OF hours-worked-rec
                THEN MOVE 2 to compare-ind
        ELSE MOVE 1 TO compare-ind.
wage-eq-hours-proc.
        PERFORM produce-output.
        PERFORM read-wage.
        PERFORM read-hours.
wage-lt-hours-proc.
        PERFORM read-wage.
wage-gt-hours-proc.
        PERFORM read-hours.
produce-output.
        MOVE employee-number OF hourly-wage-rec
          TO employee-number OF gross-pay-rec.
        MULTIPLY hourly-wage BY hours-worked GIVING gross-pay.
        ADD gross-pay TO total-gross-pay.
        WRITE gross-pay-rec.
end-of-job SECTION.
        CLOSE hourly-wage-in.
        CLOSE hours-worked-in.
        CLOSE gross-pay-out.
        OPEN OUTPUT total-gross-pay-out.
        WRITE total-gross-pay-rec.
        CLOSE total-gross-pay-out.
        STOP RUN.
```

Fig. 5. HIBOL Program for PAYROLL2

```
DATA DIVISION

KEY SECTION
          KEY EMPLOYEE-NUMBER
              FIELD TYPE IS NUMBER
              FIELD LENGTH IS 9

INPUT SECTION
          FILE HOURLY-WAGE
               KEY IS EMPLOYEE-NUMBER
          FILE HOURS-WORKED
               KEY IS EMPLOYEE-NUMBER

OUTPUT SECTION
          FILE TOTAL-GROSS-PAY
          FILE GROSS-PAY
          KEY IS EMPLOYEE-NUMBER

COMPUTATION DIVISION

TOTAL-GROSS-PAY IS (SUM OF (HOURLY-WAGE * HOURS-WORKED))

GROSS-PAY IS (HOURLY-WAGE * HOURS-WORKED)
```

Fig. 6. COBOL Program for DBINIT

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT CRADATE ASSIGN TO UT-S-LCRADATE.
        SELECT CRADB ASSIGN TO DA-I-LCRADB
        ACCESS IS SEQUENTIAL
        RECORD KEY IS CRADB-RECORD-KEY.
DATA DIVISION.
FILE SECTION.
FD  CRADATE
    LABEL RECORD IS OMITTED
    DATA RECORD IS DATEREC.
01  DATEREC.
    03  BILLING-PERIOD PICTURE X.
          B8  BEGINNING-NEW-PERIOD VALUE '1'.
FD  CRADB
    LABEL RECORDS ARE STANDARD
    RECORDING MODE IS F
    BLOCK CONTAINS 0 RECORDS
    RECORD CONTAINS 44 CHARACTERS
    DATA RECORD IS DBREC.
01  CRADBREC.
    03  DELETE-OR-DATE-INDICATOR PICTURE X.
        88  DATE-RECORD VALUE 'D'.
        B8  RECORD-DELETED VALUE HIGH-VALUE.
    03  CRADB-RECORD-KEY.
        05  CRADB-DEPARTMENT PICTURE XX.
        05  CRADB-EMP-CLASS PICTURE XX.
        05  CRADB-EMP-NO PICTURE X(5).
    03  CRADB-YTD-HOURS PICTURE S9(4)V9.
    03  CRADB-JTD-HOURS PICTURE S9(4)V9.
    03  CRADB-WEEK-HOURS PICTURE S9(4)V9.
    03  CRADB-WEEK-LABOR-COST PICTURE S9(5)V99.
    03  CRADB-PERIOD-HOURS PICTURE S9(4)V9.
    03  CRADB-PERIOD-LABOR-COST PICTURE S9(5)V99.
WORKING-STORAGE SECTION.
77  END-OF-CRADB-INDICATOR PICTURE S9 VALUE ZERO.
    B8  END-OF-CRADB VALUE 1.
77  END-CRADATE-INDICATOR PICTURE S9 VALUE ZERO.
    BB  NO-CRADATE VALUE 1.

PROCEDURE DIVISION.
initialization SECTION.
    OPEN INPUT CRADATE.
    OPEN I-O CRADB.
    READ CRADATE AT END MOVE +1 TO END-CRADATE-INDICATOR.
    IF NO-CRADATE THEN NEXT SENTENCE
                  ELSE PERFORM control-010 UNTIL END-OF-CRADB.
    CLOSE CRADATE CRADB.
    STOP RUN.
control-010.
    PERFORM read-cradb-020.
    IF END-OF-CRADB THEN NEXT SENTENCE
                    ELSE PERFORM initialize-030
                         PERFORM rewrite-040.
read-cradb-020.
    READ CRADB NEXT RECORD AT END MOVE +1 TO END-OF-CRADB-INDICATOR.
initialize-030.
    MOVE ZEROES TO CRADB-WEEK-HOURS CRADB-WEEK-LABOR-COST.
    IF BEGINNING-NEW-PERIOD
       THEN MOVE ZEROS TO CRADB-PERIOD-HOURS
                          CRADB-PERIOD-LABOR-COST.
rewrite-040.
    REWRITE CRADBREC.
```

Fig. 7. HIBOL Program for DBINIT

```
DATA DIVISIDN
KEY SECTION
          KEY CRADB-EMP-NO
              FIELD TYPE IS STRING
              FIELD LENGTH IS 5
          KEY CRADB-EMP-CLASS
              FIELD TYPE IS STRING
              FIELD LENGTH IS 2
          KEY CRADB-DEPARTMENT
              FIELD TYPE IS STRING
              FIELD LENGTH IS 2
INPUT SECTIDN
          FILE BILLING-PERIOD
          FILE DELETE-DR-DATE-INDICATOR
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-YTD-HOURS
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-JTD-HOURS
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-WEEK-HOURS
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-WEEK-LABOR-COST
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-PERIOD-HOURS
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-PERIDD-LABOR-COST
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
OUTPUT SECTION
          FILE DELETE-OR-DATE-INDICATOR
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-YTD-HOURS
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-JTD-HOURS
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-WEEK-HOURS
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-WEEK-LABOR-COST
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-PERIOD-HOURS
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
          FILE CRADB-PERIOD-LABOR-COST
               KEYS ARE CRADB-DEPARTMENT CRADB-EMP-CLASS CRADB-EMP-NO
COMPUTATIDN DIVISION

DELETE-OR-DATE-INDICATOR IS DELETE-OR-DATE-INDICATOR IF (BILLING-PERIOD PRESENT)

CRADB-YTD-HOURS IS CRADB-YTD-HOURS IF (BILLING-PERIOD PRESENT)

CRADB-JTD-HOURS IS CRADB-JTD-HOURS IF (BILLING-PERIOD PRESENT)

CRADB-WEEK-HOURS IS 0. IF ((BILLING-PERIOD PRESENT) AND
                           ((LAST PERIOD'S CRADB-WEEK-HOURS) PRESENT))

CRADB-WEEK-LABOR-COST IS 0. IF ((BILLING-PERIOD PRESENT) AND
                                ((LAST PERIOD'S CRADB-WEEK-LABOR-COST) PRESENT))

CRADB-PERIOD-HOURS IS
  CRADB-PERIOD-HOURS IF (NOT (BILLING-PERIOD = "1"))
             ELSE 0. IF ((BILLING-PERIOD = "1") AND
                         ((LAST PERIOD'S CRADB-PERIOD-HOURS) PRESENT))

CRADB-PERIOD-LABOR-COST IS
  CRADB-PERIOD-LABOR-COST IF (NOT (BILLING-PERIDD = "1"))
                  ELSE 0. IF ((BILLING-PERIOD = "1") AND
                              ((LAST PERIOD'S CRADB-PERIOD-LABOR-COST) PRESENT))
```

Fig. 8. COBOL Program for LOC-LIST

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT LIB-IN ASSIGN TO LOCIN.
        SELECT LIB-TRANS ASSIGN TO LOCTRANS.
        SELECT LIB-OUT ASSIGN TO LOCOUT.
DATA DIVISION.
FILE SECTION.
FD  LIB-IN
        LABEL RECORDS ARE OMITTED
        DATA RECORDS ARE LIBIN-REC.
01  LIBIN-REC.
        02  IN-REC.
        03  LOCATION-ONE PICTURE 99.
        03  LOCATION-TWO PICTURE 99.
        03  LIB-BUILDING-NAME PICTURE X(35).
FD  LIB-OUT
        LABEL RECORDS ARE OMITTED
        DATA RECORDS ARE LIBOUT-REC.
01  LIBOUT-REC.
        02  OUT-REC.
        03  LOCATION-ONE PICTURE 99.
        03  LOCATION-TWO PICTURE 99.
        03  BUILDING-NAME PICTURE X(35).
FD  LIB-TRANS
        LABEL RECORDS ARE OMITTED
        DATA RECORDS ARE LIBTRANS-REC.
01  LIBTRANS-REC.
        02  DELETE-IND-IN PICTURE X.
        02  TRANS-REC.
        03  LOCATION-ONE PICTURE 99.
        03  LOCATION-TWO PICTURE 99.
        03  TRANS-BUILDING-NAME PICTURE X(35).
WORKING-STORAGE SECTION.
77  DELETE-IND PICTURE X VALUE SPACE.
        88  DELETE-FLAG VALUE 'D'.
77  END-OF-LIB-IND PICTURE 9 VALUE ZERO.
        88  END-OF-LIB VALUE 1.
77  END-OF-TRANS-IND PICTURE 9 VALUE ZERO.
        88  END-OF-TRANS VALUE 1.
77  COMPARE-IND PICTURE 9 VALUE ZERO.
        88  TRANS-EQ-LIB VALUE 1.
        88  TRANS-LT-LIB VALUE 2.
        88  TRANS-GT-LIB VALUE 3.
PROCEDURE DIVISION.
HOUSEKEEPING SECTION.
        OPEN INPUT LIB-IN LIB-TRANS
             OUTPUT LIB-OUT.
        PERFORM READ-TRANSACTION.
        PERFORM READ-LIBRARY.
MAINLINE.
        IF END-OF-TRANS THEN GO TO FINISH-LIB.
        IF END-OF-LIB THEN GO TO FINISH-TRANS.
        PERFORM COMPARE.
        IF TRANS-EQ-LIB THEN PERFORM TRANS-EQ-LIB-PROC.
        IF TRANS-LT-LIB THEN PERFORM TRANS-LT-LIB-PROC.
        IF TRANS-GT-LIB THEN PERFORM TRANS-GT-LIB-PROC.
        PERFORM PRODUCE-OUTPUT.
        GO TO MAINLINE.
READ-LIBRARY.
        READ LIB-IN AT END MOVE 1 TO END-OF-LIB-IND.
READ-TRANSACTION.
        READ LIB-TRANS AT END MOVE 1 TO END-OF-TRANS-IND.
```

Fig. 8. COBOL Program for LOC-LIST (CONTINUED)

```
COMPARE.
          IF LOCATION-ONE OF TRANS-REC > LOCATION-ONE OF IN-REC
        THEN MOVE 3 TO COMPARE-IND
        ELSE IF LOCATION-ONE OF TRANS-REC < LOCATION-ONE OF IN-REC
        THEN MOVE 2 TO COMPARE-IND
        ELSE IF LOCATION-TWO OF TRANS-REC > LOCATION-TWO OF IN-REC
        THEN MOVE 3 TO COMPARE-IND
        ELSE IF LOCATION-TWO OF TRANS-REC < LOCATION-TWO OF IN-REC
        THEN MOVE 2 TO COMPARE-IND
        ELSE MOVE 1 TO COMPARE-IND.
TRANS-EQ-LIB-PROC.
        MOVE DELETE-IND-IN TO DELETE-IND.
        MOVE TRANS-REC TO OUT-REC.
        PERFORM READ-LIBRARY.
        PERFORM READ-TRANSACTION.
TRANS-LT-LIB-PROC.
        MOVE DELETE-IND-IN TO DELETE-IND.
        MOVE TRANS-REC TO OUT-REC.
        PERFORM READ-TRANSACTION.
TRANS-GT-LIB-PROC.
        MOVE IN-REC TO OUT-REC.
        PERFORM READ-LIBRARY.
PRODUCE-OUTPUT.
        IF NOT DELETE-FLAG THEN WRITE LIBOUT-REC.
        MOVE SPACE TO DELETE-IND.
FINISH-LIB.
        IF END-OF-LIB THEN GO TO EOJ.
        MOVE IN-REC TO OUT-REC.
        PERFORM PRODUCE-OUTPUT.
        PERFORM READ-LIBRARY.
        GO TO FINISH-LIB.
FINISH-TRANS.
        IF END-OF-TRANS THEN GO TO EOJ.
        MOVE DELETE-IND-IN TO DELETE-IND.
        MOVE TRANS-REC TO OUT-REC.
        PERFORM PRODUCE-OUTPUT.
        PERFORM READ-TRANSACTION.
        GO TO FINISH-TRANS.
EOJ.
        CLOSE LIB-IN LIB-TRANS LIB-OUT.
        STOP RUN.
```

Fig. 9. HIBOL Program for LOC-LIST

```
DATA DIVISION
KEY SECTION
          KEY LOCATION-ONE
              FIELD TYPE IS NUMBER
              FIELD LENGTH IS 2
INPUT SECTION
          FILE LIB-BUILDING-NAME
               KEY IS LOCATION-ONE
          FILE DELETE-IND-IN
               KEY IS LOCATION-ONE
          FILE TRANS-BUILDING-NAME
               KEY IS LOCATION-ONE
OUTPUT SECTION
          FILE BUILDING-NAME
               KEY IS LOCATION-ONE
COMPUTATION DIVISION
BUILDING-NAME IS   LIB-BUILDING-NAME IF (NOT (TRANS-BUILDING-NAME PRESENT))
            ELSE TRANS-BUILDING-NAME IF (NOT (DELETE-IND-IN = "D"))
```

1.5 Outline of Remaining Chapters

The remainder of this document is broken into six chapters. Chapters 2 and 3 give brief introductions to COBOL and HIBOL, respectively. Chapter 4 provides an in-depth description of analyzed plans. Chapter 5 discusses the current implementation of the portion of the system that translates the analyzed plans into HIBOL. Chapter 6 discusses possible methods of improving and expanding the translation process. Chapter 7 discusses the entire system from a more global perspective and suggests directions for further research.

2. COBOL

Since COBOL is a very widely known language, and references for COBOL abound, this chapter will give the briefest possible introduction of those features of COBOL that must be understood by the reader in order to comprehend the remainder of this document. Readers familiar with COBOL are invited to skip all but the first section of this chapter. Readers who want further information about COBOL are referred to [22] and [11].

COBOL (COmmon Business Oriented Language) is a high level programming language designed by the Conference On DAta SYstems Languages (CODASYL) for use in data processing tasks. It now has several standard versions supported by the American National Standard Institute (ANSI). The COBOL syntax used in this document does not exactly match any of the ANSI standards or any particular vendor's syntax, all of which vary in minor ways. Instead, it follows a common subset, and certain esoteric syntax requirements are ignored altogether.

2.1 Example Programs

A sample COBOL program is shown in Figure 10. This program, entitled "PAYROLL", will be used for many examples throughout this document, and therefore should be understood by the reader. To this end, a short discussion of the function performed by this program is appropriate.

PAYROLL is a relatively trivial program which might appear in a simple payroll system. It uses a single input file called "HOURLY-WAGE-IN". HOURLY-WAGE-IN contains two fields, "EMPLOYEE-NUMBER" and "HOURLY-WAGE". EMPLOYEE-NUMBER is the key field for this file. It is a nine digit social security number that is used to specify which employee a given record in the file is associated with. HOURLY-WAGE is the single data field that specifies the hourly wage earned by the corresponding employee.

PAYROLL produces three output files. The first of these, GROSS-PAY-OUT, contains a record for each record contained in HOURLY-WAGE-IN. GROSS-PAY-OUT has two fields: EMPLOYEE-NUMBER and GROSS-PAY. EMPLOYEE-NUMBER is again the key field. GROSS-PAY is a data field that contains the weekly gross pay earned by the employee. The program assumes all employees work forty hours per week. The other two output files, EMPLOYEE-COUNT-OUT and TOTAL-GROSS-PAY-OUT, each contain only a single record. Therefore, they have no key fields. EMPLOYEE-COUNT-OUT has a single data field, EMPLOYEE-COUNT, which contains the number of employee records processed by the program. TOTAL-GROSS-PAY-OUT also has a single data field, TOTAL-GROSS-PAY, which contains the total gross pay earned by all the employees whose records are processed by the program.

Fig. 10. COBOL Program for PAYROLL

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PAYROLL.
AUTHOR.             G. FAUST.
INSTALLATION.       PDP10.
DATE-WRITTEN.       2/20/80.
DATE-COMPILED.      NEVER.


ENVIRONMENT DIVISION,
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
      SELECT HOURLY-WAGE-IN ASSIGN TO DA-2301-S-HWI.
      SELECT GROSS-PAY-OUT ASSIGN TO DA-2301-S-GPO.
      SELECT EMPLOYEE-COUNT-OUT ASSIGN TO DA-2301-S-ECO.
      SELECT TOTAL-GROSS-PAY-OUT ASSIGN TO DA-2301-S-TGPO.


DATA DIVISION.
FILE SECTION.

FD  hourly-wage-in
    LABEL RECORD IS OMITTED
    DATA RECORD IS hourly-wage-rec.
01  hourly-wage-rec.
    02  employee-number                        PICTURE IS 9(9).
    02  hourly-wage                            PICTURE IS 999V99.

FD  gross-pay-out
    LABEL RECORD IS OMITTED
    DATA RECORD IS gross-pay-rec.
01  gross-pay-rec.
    02  employee-number                        PICTURE IS 9(9).
    02  gross-pay                              PICTURE IS 999V99.

FD  employee-count-out
    LABEL RECORD IS OMITTED
    DATA RECORD IS employee-count-rec.
01  employee-count-rec.
    02  employee-count                         PICTURE IS 9(6).

FD  total-gross-pay-out
    LABEL RECORD IS OMITTED
    DATA RECORD IS total-gross-pay-rec.
01  total-gross-pay-rec.
    02  total-gross-pay                        PICTURE IS 9(7)V99.
```

Fig. 10. COBOL Program for PAYROLL (CONTINUED)

```
PROCEDURE DIVISION.
initialization SECTION.
       MOVE ZERO TO total-gross-pay.
       MOVE ZERO TO employee-count.
       OPEN INPUT hourly-wage-in.
       OPEN OUTPUT gross-pay-out.
mainline SECTION.
       READ hourly-wage-in AT END GO TO end-of-job.
       MOVE employee-number OF hourly-wage-rec
         TO employee-number OF gross-pay-rec.
       MULTIPLY hourly-wage BY 40 GIVING gross-pay.
       ADD 1 TO employee-count.
       ADD gross-pay TO total-gross-pay.
       WRITE gross-pay-rec.
       GO TO mainline.
end-of-job SECTION.
       CLOSE hourly-wage-in.
       CLOSE gross-pay-out.
       OPEN OUTPUT employee-count-out.
       WRITE employee-count-rec.
       CLOSE employee-count-out.
       OPEN OUTPUT total-gross-pay-out.
       WRITE total-gross-pay-rec.
       CLOSE total-gross-pay-out.
       STOP RUN.
```

COBOL programs for the other examples used in this document (PAYROLL2, DBINIT, and LOC-LIST) can be found in Section 1.4. The second example, PAYROLL2, is an expanded version of PAYROLL which eliminates the assumption that every employee works forty hours a week. Instead, HOURS-WORKED, a data field in the HOURS-WORKED-IN file, is used in the computation of GROSS-PAY. PAYROLL2 is an important test case because it includes a computation that uses data fields from two different files.

The third program, DBINIT, is a simple data base initialization program which uses two input files. The first of these files, CRADATE, has only a single record with a single data field. This singleton piece of information, called "BILLING-PERIOD", controls the initialization of certain data fields in the second file. The second file, CRADB, is an indexed file that is accessed sequentially. Note that the program does nothing at all if CRADATE is initially empty; i.e. if the value of BILLING-PERIOD is unknown. This program was included because of its use of REWRITE to perform a file update operation.

The fourth program, LOC-LIST is an example of a file update program using a transaction file. The first input file, LIB-IN, is a library file containing building names associated with location code key fields. The second input file, LIB-TRANS, is the transaction file used to update the library. The updated library is output into the only output file, LIB-OUT. The updated library will contain a record for every set of key values that appears in only one of the two input files. In addition, if a set of key values appears in both input files, then the data values in the updated library file are taken from the transaction file, except when the first field of the transaction file, called "DELETE-IND-IN", contains a "D" (mnemonic for delete) in which case no record will appear in the updated library file for that set of key values. This program is an important example because it performs a file merge operation.

As can be seen in Figure 10, a COBOL program is broken up into four main divisions; IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. The IDENTIFICATION DIVISION is primarily for documentation of the program, and contains no information that is pertinent to the current discussion. The only information that is contained in the ENVIRONMENT DIVISION that is pertinent is the information concerning file organizations and file access methods. (None of this information is shown in Figure 10 because all files accessed by this program take the default value for these two parameters.) The DATA DIVISION contains information about the structure of files in terms of the data fields that comprise a record in those files, as well as information about all other variables used within the program. The PROCEDURE DIVISION contains a procedural representation of the algorithm used to implement the desired computation.

2.2 ENVIRONMENT DIVISION

For the purposes of this document, there are two possible organizations for a file that is used within a COBOL program; sequential and indexed. A sequential file can either originate from a magnetic tape, or from a random access device such as a magnetic disk. In either case, the feature of a file that makes it a sequential file is that the records in that file are stored (or can be viewed as being stored) in contiguous locations on a memory device. Whether for input or output, they can only be processed in that order.

An indexed file is broken into two components; the data itself, and an indexed list of pointers into that data. How each of these components is actually stored on a memory device is not important. The important point is that the data can be accessed sequentially, as is done with a sequential file, or randomly using an index to point to a particular record. The method used to access records in an indexed file is, appropriately, called the "access method".

The file organization for each file that is accessed by a COBOL program is specified in the ENVIRONMENT DIVISION, with the default being sequential organization. In addition, if the file is specified to be organized as an indexed file, the RECORD KEY must be specified as well as the access method for that file. The RECORD KEY is used to specify the portion of the record structure that contains the key fields for that file. (The record structure for a file will be discussed below). If the access method for an indexed file is random access, the NOMINAL KEY must also be specified. The NOMINAL KEY is a storage area used in the PROCEDURE DIVISION of the program to contain the index which specifies the location in the file that should be accessed by the next INPUT/OUTPUT operation.

2.3 DATA DIVISION

The DATA DIVISION of a COBOL program is broken into two sections; FILE and WORKING-STORAGE. The FILE SECTION contains, for each file that will be accessed by the program, information about the structure of an individual record within that file. The WORKING-STORAGE SECTION contains information about all other variables and storage areas used during the execution of the program.

Associated with each file accessed by a COBOL program is a buffer area. All INPUT/OUTPUT operations performed on a file place information In, or take information from, that buffer area. The buffer area for a file is given a structure, called the "buffer-structure" or "record structure", in the FILE SECTION which specifies the fields that are contained within a record in that file. Definitions of the record structures for the files accessed by PAYROLL are shown in Figure 10. A record structure can be an arbitrary tree structure. The level of a particular structure element is indicated by the number that precedes the name given to that structure element. Lower numbers are closer to the root of the tree. For example, consider the structure definition for the buffer associated with HOURLY-WAGE-IN. HOURLY-WAGE-REC is the name given to the structure element that corresponds to the entire buffer area. The leaves of the tree are the individual fields in the file. In this example, they are EMPLOYEE-NUMBER and HOURLY-WAGE.

Fig. 11. PICTURE Elements

PICTURE ELEMENT	MEANING
9	Decimal Digit
V	Implied Decimal Point
X	Any ASCII Character
(*num*)	Repeat Count: The preceding PICTURE element is repeated *num* times.

Each leaf element in the structure is given a PICTURE clause. The PICTURE clause specifies the data type and length of the particular field by giving a picture of the typical value stored in that field. Figure 11 gives a list of common picture elements and their meaning. An examination of Figures 10 and 11 will reveal, lor example, that EMPLOYEE-NUMBER is a nine digit integer and HOURLY-WAGE is a five digit number with two of the digits to the right of the decimal point.

The WORKING-STORAGE SECTION defines all data areas used during the execution of the program except those corresponding to file buffer areas. Data areas defined in WORKING-STORAGE can have tree structures exactly like the structures associated with file buffers. In addition, there are two variable types that are unique to WORKING-STORAGE: 77 variables, and 88 variables (so named because of the numbers used to designate them). A 77 variable is a simple variable with no structure whose type and length is specified in a PICTURE clause.

An 88 variable is used to set up a flag. It does not define an additional storage area, but provides a named way to refer to a predicate; one that decides whether or not a given area contains a particular value. For example, consider the portion of WORKING-STORAGE defined in Figure 12. Initially, the COMPARE-FLAG-AREA contains a 0, so specified by the VALUE clause which can be used anywhere within the WORKING-STORAGE SECTION to initialize storage areas. The two 88 variables, NEGATIVE and NON-NEGATIVE, are used in the PROCEDURE DIVISION to test if that area currently contains a 1 or a 2. When the area contains a 1, NEGATIVE will evaluate to TRUE. Otherwise it will evaluate to FALSE. Similarly, NON-NEGATIVE will evaluate to TRUE when the area contains a 2, and to FALSE otherwise. Initially, they will both evaluate to FALSE (since the area contains a 0) and will continue to do so until a 1 or 2 is moved into COMPARE-FLAG-AREA. In order to make all this work, a COBOL program that includes the definition of COMPARE-FLAG-AREA shown in Figure 12 may well contain a statement of the form

```
IF variable < 0 THEN MOVE 1 TO COMPARE-FLAG-AREA
                ELSE MOVE 2 TO COMPARE-FLAG-AREA.
```

somewhere within the PROCEDURE DIVISION. The reader should note that the inclusion of 88 variables in COBOL makes using flags trivial, and they will, therefore, appear often in COBOL programs. Any system that hopes to translate COBOL programs needs to be able to handle flags in a reasonable fashion.

Fig. 12. Example Use of 88 Variables

```
77  COMPARE-FLAG-AREA PICTURE 9 VALUE ZERO.
    88  NEGATIVE VALUE 1.
    88  NON-NEGATIVE VALUE 2.
```

2.4 PROCEDURE DIVISION

The PROCEDURE DIVISION contains a procedural representation of the particular algorithm used to implement the desired computation. For the purposes of this document, it is only necessary to understand a small subset of the possible statement forms that can appear in the PROCEDURE DIVISION.

A variable name used in the PROCEDURE DIVISION must provide a unique reference to a data storage area. Names that correspond to 77 and 88 variables must always be unique. Names that refer to substructures, however, may not be unique. To eliminate this difficulty, COBOL supplies the OF clause to be used in references to data areas in structures. For example, suppose that two structures both contain substructure data areas associated with the same name. Then a unique reference to the substructure area in the first structure is:

```
substructure-name OF structure-name-1
```

while a unique reference to the substructure area in the second structure is:

```
substructure-name OF structure-name-2
```

There are four main statements that affect control flow within a COBOL program; STOP RUN, GO TO, IF-THEN-ELSE, and PERFORM. Three of them are very simple and are shown in Figure 13. (The square brackets are used to signify an optional parameter). The STOP RUN statement terminates execution of the program. The COBOL GO TO and IF-THEN-ELSE constructs are no different from their counterparts used in procedural languages in general and need no further explanation.

Fig. 13. Simple Statements that Affect Control Flow

```
STOP RUN

GO TO label

IF predicate THEN imperative-statement-1
          [ELSE imperative-statement-2]
```

Fig. 14. PERFORM Syntax when Used to Implement a Subroutine Call

```
PERFORM paragraph-one [THROUGH paragraph-two]
```

The PERFORM statement, however, is unique to COBOL. It is used to implement two different constructs: a loop construct and a weak form of subroutine call. The syntax of the PERFORM statement when used as a subroutine call is shown in Figure 14. In COBOL, a paragraph is all the code starting at a label, which is used as the paragraph name, and continuing up to but not including the next label. The PERFORM statement in Figure 14 indicates that control should be passed to the label signified by *paragraph-one* and that processing will continue either to the end of that paragraph, or to the end of *paragraph-two* if the optional THROUGH clause is used. In either case, control is returned to the statement following the PERFORM after the above stated processing is completed. This is a weak form of subroutine call because no arguments are passed; the paragraphs that are processed use only global values and recursion is not allowed.

Used as a loop construct, the PERFORM statement has three possible forms as shown in Figure 15. These forms should be self explanatory. Note that these forms allow for both the indexed loop construct and the DO-WHILE construct.

COBOL has a number of statements used to manipulate data. The data manipulation statements used in this document are shown in Figure 16. In the MULTIPLY and DIVIDE statements using the BY clause, if the GIVING clause is omitted the result of the operation is placed in *operand-1*. If the GIVING clause is included, both operands remain as they were, and the result is placed into *result*. The DIVIDE statement using the INTO clause is the same as the DIVIDE statement using the BY clause except that the operands are reversed. In the ADD and SUBTRACT

Fig. 15. PERFORM Syntax when Used as a Loop Construct

```
PERFORM paragraph-one [THROUGH paragraph-two] integer TIMES

PERFORM paragraph-one [THROUGH paragraph-two] UNTIL predicate

PERFORM paragraph-one [THROUGH paragraph-two]
        VARYING variable FROM integer-1 TO integer-2
        BY integer-3 UNTIL predicate
```

Fig. 16. Data Manipulation Statements

```
MULTIPLY operand-1 BY operand-2 [GIVING result]
DIVIDE operand-1 BY operand-2 [GIVING result]
DIVIDE operand-1 INTO operand-2 [GIVING result]
ADD operand-1 TO operand-2 [GIVING result]
SUBTRACT operand-1 FROM operand-2 [GIVING result]
MOVE source TO destination
```

statements, if the GIVING clause is omitted, the result is placed into `operand-2`. If the GIVING clause is included, both operands remain as they were and the result is placed into `result`. The MOVE statement is used to move information from one data area into another.

Statements used to manipulate files are shown in Figure 17. The OPEN statement is used to prepare files to be accessed. There are three possible access types; INPUT, OUTPUT, and I-O. A file opened for INPUT is read only. A file opened for OUTPUT is write only. A file opened for I-O can be read from and written to. The CLOSE statement is used to release a file when it is no longer needed.

The three different forms of the READ statement are used to access information in different types of files. The first form is used to access files that have a sequential file organization. The second form is used to access files that have an indexed file organization when the access method is random access. The third form is used to access files that have an indexed file organization when the access method is sequential access. The AT END and INVALID KEY clauses specify that the `imperative-statement` should be performed when the requested record cannot be read from the file.

The WRITE statement is used to place information into a file. It can be used on any of the file types. When applied to a sequential file, the WRITE statement always appends records to the end of the file, When used on an indexed file accessed randomly, it writes a record at the place in the file designated by the NOMINAL KEY. When used on an indexed file accessed sequentially, it writes over the record most recently read. The REWRITE statement can only be used on files opened with an `io-type` of I-O. It always writes over the record most recently accessed. Note that (for esoteric reasons not discussed here) a READ statement takes a file-name as its argument while a WRITE or REWRITE statement takes a file-buffer-structure-name as its argument.

Fig. 17. File Manipulation Statements

```
OPEN  io-type file-name-1 [file-name-2 . . .]
     [io-type file-name-3 [file-name-4 . . .]] . . .
Where io-type is one of: INPUT, OUTPUT, or I-O

CLOSE file-name-1 [file-name-2 . . .]

READ file-name AT END imperative-statement
READ file-name INVALID KEY imperative-statement
READ file-name NEXT RECORD AT END imperative-statement

WRITE file-buffer-structure-name
REWRITE file-buffer-structure-name [INVALID KEY imperative-statement]
```

3. HIBOL

HIBOL is a very high level single assignment programming language designed for expressing data processing application programs in such a way that the form of the program closely resembles functional specifications for the application. It is intended to be automatically translated into a conventional high level language such as PL/1 or COBOL via an automatic programming system called PROTOSYSTEM I [27]. It is a descriptive rather than a procedural language; the exact procedures used to effect the actual processing are not explicitly represented. A HIBOL program for the PAYROLL example is presented in Figure 18.

The kernel idea for PROTOSYSTEM I was initially conceived by William Martin [17]. Martin and Ruth [27,18,8,28] then developed PROTOSYSTEM I (which produces compilable PL/1 programs and the necessary IBM JCL from HIBOL) with the help of others: most notably Baron, Burke, Kornfeld, Morgenstern, and Thomas [3,14,21,30].

HIBOL can be viewed as a language in which algorithms are expressed in terms of computations performed on streams. It is important to keep this viewpoint in mind for two reasons. First, it will aid in the understanding of HIBOL primitives and how they interact. Second, it will be used in a later chapter to relate HIBOL to other programming languages.

The basic elements of description of a data processing application can be broken into two categories: those that describe data and those that describe operations performed on that data. In HIBOL, the descriptive elements are correspondingly divided into a DATA DIVISION and a COMPUTATION DIVISION. The next two sections of this chapter are similarly divided.

3.1 DATA DIVISION

HIBOL uses a single data type called a "flow". A flow is a set of related data items each of which is associated with a unique multi-component index. Each index component is called a "key". The set of all possible sets of values for the keys of a particular flow is called the "universal key space" of that flow. The set of sets of key values that actually appear in a given instance of a flow is called the "actual key space" for that instance of the flow. For example, if a flow has a single key that is a four digit integer representing a client identification number, then the cardinality of the universal key space for that flow is 10,000, while the cardinality of the actual key space for that flow is the number of clients that actually exist and might be as low as zero or as high as 10,000.

Each element of a flow has a set of key values and a single data value. The typical data processing concept of a file record containing a set of key values and multiple data values (such as a COBOL file record) is abstracted in HIBOL as separate flow elements from different flows, all of which have the same set of key values, and each of which has one of the data values. This method of describing the organization of sets of data values disassociates the logical organization of the data from the physical organization of the data; the semantics of HIBOL describe the logical organization while leaving the physical organization unspecified.

A named flow is called a "data-set". Data-sets are divided into three categories; input, output, and variable. Input and output data-sets define the inputs to and outputs from the computation represented by a HIBOL program. The variable data-sets are used for intermediate values formed in the computation.

The DATA DIVISION at the top of the HIBOL program for PAYROLL shown in Figure 18 gives an example of the specification of data-sets. The first part of the DATA DIVISION is the KEY SECTION. In this section, each key that is going to be used in the specification of any of the data-sets must be specified along with its field type and length. In this example, EMPLOYEE-NUMBER is the sole key and is an integer with a field length of nine (a social security number).

The next two sections of the DATA DIVISION specify the input and output data-sets that are going to be used in the program (see Figure 18). Each data-set specification is preceded by the keyword "FILE". The HOURLY-WAGE, and GROSS-PAY data-sets both use the key EMPLOYEE-NUMBER, while TOTAL-GROSS-PAY and EMPLOYEE-COUNT do not have any key at all. In this latter case, the cardinality of the universal key space is one, and the actual key space will contain at most a singleton value. If the PAYROLL example used any variable data-sets, a VARIABLE SECTION, identical in format to the INPUT and OUTPUT sections, would appear in the DATA DIVISION right after the INPUT SECTION.

3.2 COMPUTATION DIVISION

Following the DATA DIVISION is the COMPUTATION DIVISION. The COMPUTATION DIVISION contains a *single* definition for each output and variable data-set. Each data-set definition is of the form

```
data-set-name IS flow-expression
```

The flow expression on the right hand side of a data-set definition must have the same universal key space as the data-set referred to by the name on the left hand side. The semantics of a flow expression dictate that there is an implicit iteration over all values of the actual key space of the flow represented by that expression.

Fig. 18. HIBOL Program for PAYROLL

```
DATA DIVISION

KEY SECTION

          KEY EMPLOYEE-NUMBER
          FIELD TYPE IS NUMBER
          FIELD LENGTH IS 9

INPUT SECTION

          FILE HOURLY-WAGE
          KEY IS EMPLOYEE-NUMBER

OUTPUT SECTION

          FILE GROSS-PAY
          KEY IS EMPLOYEE-NUMBER

          FILE EMPLOYEE-COUNT

          FILE TOTAL-GROSS-PAY

COMPUTATION DIVISION

TOTAL-GROSS-PAY IS (SUM OF (HOURLY-WAGE * 40.))

EMPLOYEE-COUNT IS (COUNT OF HOURLY-WAGE)

GROSS-PAY IS (HOURLY-WAGE * 40.)
```

Fig. 19. HIBOL Syntax for Conditional Expressions

```
data-set-name IS       flow-expression-1 IF predicate-1
                 [ELSE flow-expression-2 IF predicate-2]. . .
                 [ELSE flow-expression-n]
```

There is only one statement form in HIBOL that can cause conditional computation. This statement form is shown in Figure 19. The syntax of this form resembles an IF-THEN-ELSE, but it has the semantics of a CASE construct. Since *data-set-name* can be given the value corresponding to the flow expression of any of the clauses, those flow expressions must all express flows that have the same universal key space as the data-set referred to by *data-set-name*. The conditional form is defined over the union of the actual key spaces of the flow expressions used in the clauses. When such a conditional form is evaluated for a particular index value in that union, the predicates are evaluated in order, starting with *predicate-1*. As soon as any of them evaluates to TRUE, the conditional form is given the value of the flow expression corresponding to it for that set of key values. If none of the predicates evaluates to TRUE and the optional final ELSE clause is included, the conditional form is given the value of the final flow expression. If none of the predicates evaluates to TRUE and the optional final ELSE clause is *not* included, the value of the conditional form is undefined and the corresponding index is excluded from the actual key space of the resultant flow.

For example, consider the HIBOL program fragment shown in Figure 20. In this example, the output data-set PROFITABLE-DEPARTMENT contains an element for every element in the input data-set, DEPARTMENT-BALANCE, which has a balance greater than zero. The elements in the actual key space of DEPARTMENT-BALANCE that have a balance less-than or equal to zero are excluded from the actual key space of PROFITABLE-DEPARTMENT.

Flow expressions can contain the usual arithmetic operations appearing in any programming language. The syntax for such arithmetic operators, shown in Figure 21, is exactly what one might expect. The semantics of such expressions, however, is quite different from the semantics of similar looking expressions in other languages. The two flow expressions used as operands to the arithmetic

Fig. 20. HIBOL Program Fragment with Conditional Form

```
DATA DIVISION

INPUT SECTION

       FILE DEPARTMENT-BALANCE
       KEY IS DEPARTMENT-NUMBER

OUTPUT SECTION

       FILE PROFITABLE-DEPARTMENT
       KEY IS DEPARTMENT-NUMBER

COMPUTATION DIVISION

PROFITABLE-DEPARTMENT IS DEPARTMENT-BALANCE IF DEPARTMENT-BALANCE > 0.
```

Fig. 21. HIBOL Syntax for Arithmetic Operators

```
flow-expression-1 * flow-expression-2
flow-expression-1 / flow-expression-2
flow-expression-1 + flow-expression-2
flow-expression-1 - flow-expression-2
```

operators must have the same universal key space. In the case of the multiplicative operators, the actual key space of the resultant flow is the intersection of the actual key spaces of the two operands. In the case of the additive operators, the actual key space of the resultant flow is the union of the actual key spaces of the operands. To understand this in more detail, the concept of a PRESENT predicate must be introduced.

A PRESENT predicate, applied to a flow, evaluates to TRUE for all index values that are elements of the actual key space of that flow, and to FALSE for all other possible index values for that flow. So, for example, the predicate

```
HOURLY-WAGE PRESENT
```

is TRUE for all values of the key (EMPLOYEE-NUMBER) which correspond to actual employees, and FALSE for all other possible employee numbers.

Returning to the discussion of arithmetic operators, the semantics of flow expressions involving arithmetic operators are easier to understand in the form into which they are expanded by the automatic programming system. Examples are shown in Figure 22. It should be clear that the expanded expressions do produce the desired intersection and union of the actual key spaces. In either case, elements in the resultant flow are given key values that correspond to the key values of the elements in the operand flows from which they are produced.

Arithmetic operators can be used with operand flows that are not simply data-sets. In a case in which one of the operand flows is a constant, the resultant flow has the same actual key space as the non-constant operand flow. In a case in which either of the operand flows is some flow

Fig. 22. Expanded Forms of Arithmetic Flow Expressions

```
data-set-name-1 IS data-sat-name-2 * data-set-name-3

is expanded into:

data-set-name-1 IS data-set-name-2 * data-set-name-3 IF     data-set-name-2 PRESENT
                                                        AND data-set-name-3 PRESENT


data-set-name-1 IS data-sat-name-2 + data-sat-name-3

is expanded into:

data-set-name-1 IS data-set-nama-2 + data-sat-name-3 IF     data-set-name-2 PRESENT
                                                        AND data-set-name-3 PRESENT

                   data-set-name-2                   IF     data-set-name-2 PRESENT

                   data-set-name-3                   IF     data-set-name-3 PRESENT
```

expression, the semantics are just as if that flow had been a data-set, although the PRESENT predicates appearing in the expanded form of the expression will be more complex because the flow expression does not have a name associated with it. An example of the use of an arithmetic operator in a flow expression appears in the definition for the GROSS-PAY data-set in Figure 18.

In addition to arithmetic operators, HIBOL programs can Include reduction operators. The reduction operators, the syntax of which is shown in Figure 23, produce resultant flows with indices composed of fewer key components. The key components of the resultant flow must be a subset of the key components of the flow used as the operand of the reduction operator. A data element in the resultant flow with a particular index derives its value from all the data elements in the operand flow with the same values for all key components in the common subset. For example, consider the HIBOL program fragment shown in Figure 24. The input data-set, CHECK-AMOUNTS, contains an element for each check written by each bank customer during one accounting period. The output data-set, CUSTOMER-TOTAL, contains an element *for each customer* that is the sum of the amounts of the checks written by that customer in that accounting period.

Fig. 23. HIBOL Syntax for Reduction Operators

```
SUM OF flow-expression
COUNT OF flow-expression
MAX OF flow-expression
MIN OF flow-expression
```

Fig. 24. Sample HIBOL Program Fragment with Reduction Operator

```
DATA DIVISION

INPUT SECTION

       FILE CHECK-AMOUNTS
       KEYS ARE CUSTOMER-NUMBER CHECK-NUMBER

OUTPUT SECTION

       FILE CUSTOMER-TOTAL
       KEY IS CUSTOMER-NUMBER

COMPUTATION DIVISION

       CUSTOMER-TOTAL IS SUM OF CHECK-AMOUNTS
```

Two other examples of the use of reduction operators can be seen in the definitions for TOTAL-GROSS-PAY and EMPLOYEE-COUNT in Figure 18. Note that in both these cases, the resultant data-sets have no key components at all, and therefore contain only a single element.

Another feature of HIBOL is that the same data-set can appear in both the INPUT and OUTPUT sections of the DATA DIVISION. This is done when the HIBOL program performs an update operation on that data-set. It must be possible in the COMPUTATION DIVISION, however, to distinguish references to the input data-set from references to the output data-set. This is done through the use of the LAST PERIOD construct. References to the input data-set use the syntax

LAST PERIOD'S *data-set-name*

while references to the output data set simply use the syntax

data-set-name

There are many other features of HIBOL, including specifications for time intervals at which certain data sets should be generated, running totals, and formatted output reports, which will not be presented here. Although the set of HIBOL statement forms described above is not exhaustive, it is sufficient for the purposes of this document. All HIBOL code that has been produced by the SATCH system to date uses only those HIBOL constructs that have already been discussed. The reader is invited to turn now to Section 1.4 where corresponding COBOL and HIBOL programs are given for four examples (including PAYROLL), before returning to a discussion of some of the more global features of the HIBOL language.

3.3 Features of HIBOL Revisited

The specifications given in the COMPUTATION DIVISION of a HIBOL program need not be ordered in any special way by the programmer. Each can be viewed as a definition of the values that will be produced for a certain data-set. The autoprogramming system, PROTOSYSTEM I, will correctly order those computations for which the ordering is important. Note that this view of computation as definition requires that any data set name can appear at most once on the left hand side of a computation specification; i.e. HIBOL is a single assignment language. Another ramification of this view is that potential concurrency of computation can be recognized by the system and could be exploited if the target code were generated for a parallel hardware configuration.

Perhaps the most far reaching effect of this programming style is that there is no explicit notion of iteration or recursion. The only implicit iteration is that which iterates over the elements (or subsets of the elements) of an actual key space. Since HIBOL does not have explicit iteration, recursion, GOTOs, or a jump of any kind, it cannot be used to express certain computable functions in any reasonable fashion. These functions, however, rarely appear in business data processing applications, and therefore, this lack of expressive power should not be considered a major drawback. The semantics of HIBOL were designed by Hammer et. al. to express exactly the functions that appear most often in business data processing applications.

In fact, it is in just this restriction of expression that the great utility of HIBOL lies. The beauty of the language lies in the fact that the programmer is not required to worry about the details of the iterations. The necessity to deal with these details is one of the things that makes the maintenance task so difficult in other languages. In addition, the number of identifiers that the programmer has to deal with is substantially reduced, and the ones that do appear usually have a direct correspondence to some quantity in the real world. These are the features of HIBOL that make it well suited for increased programmer productivity and program clarity in the domain of data processing applications.

4. Plans and Plan Building Methods

Now that the essential features of HIBOL have been discussed, we can take a closer look at the analyzed plan from which the HIBOL is produced. While reading this chapter, it is important to consider the key features of HIBOL as we go along in order to grasp the applicability of the structure of an analyzed plan to the translation process. This chapter is meant to contain enough information to make the applicability apparent and to render the following chapters comprehensible. A much more detailed account of plans, Plan Building Methods, the process which creates a surface plan, and the process that analyzes a plan in terms of PBMs, can be found in Waters' PhD thesis [31,32].

A plan is a detailed representation of a program designed to have several useful properties. First, the plan represents the program completely; it contains all the information necessary for execution. Second, it is language independent. Therefore, it can be used to represent a program originally written in many different languages. Third, much of the information that is implicit in the program is made explicit. In particular, the control flow and data flow between different sections of the program are explicitly represented. Finally, the plan exhibits locality; features of a component of a plan can be discerned by examining information local to that component.

4.1 Surface Plans

The basic unit of a plan is a "segment". Different segments of a plan are hierarchically linked via subsegment and supersegment relations. A surface plan, an example of which is shown In Figure 25, is a representation of a program that is logically organized in much the same way as the original source language representation of the program. It has only the simplest hierarchical structure: a root segment that has all other segments in the plan as immediate inferiors. Each of the subsegments has no internal structure. They all represent primitive logical, arithmetic, or control operations.

These primitive segments, and in fact segments in general, can be placed in one of three categories, "simple", "split" or "join", depending on their interaction with control flow. A simple segment accepts control flow from exactly one place and produces control flow to exactly one other place. Examples of primitive simple segments include primitive arithmetic functions such as PLUS or TIMES, and primitive logical functions such as EQUAL or GREATER-THAN. Exactly which primitive logical and arithmetic functions can occur in a plan depends upon the source language from which the plan was built, but a standard set of primitive functions is shared by most programming languages. The library of primitive function used when the source language is COBOL is given in Appendix I.

Also included among the simple segments are constants. They differ from other simple segments in that they do not have any incoming data flow. They can be viewed as functions with no arguments that have a singleton value for their range.

A split accepts control flow from exactly one place, and produces control flow to more than one place. There are only two different primitive split segments: PIF and PIFNULL. PIF takes a single bit boolean argument and transfers control to a first segment if the boolean is TRUE and to a second segment if the boolean is FALSE. PIFNULL is simply the converse of PIF.

A join accepts control flow from more than one place and produces control flow to exactly one other place. There is only one primitive join segment. It is called "JOIN".

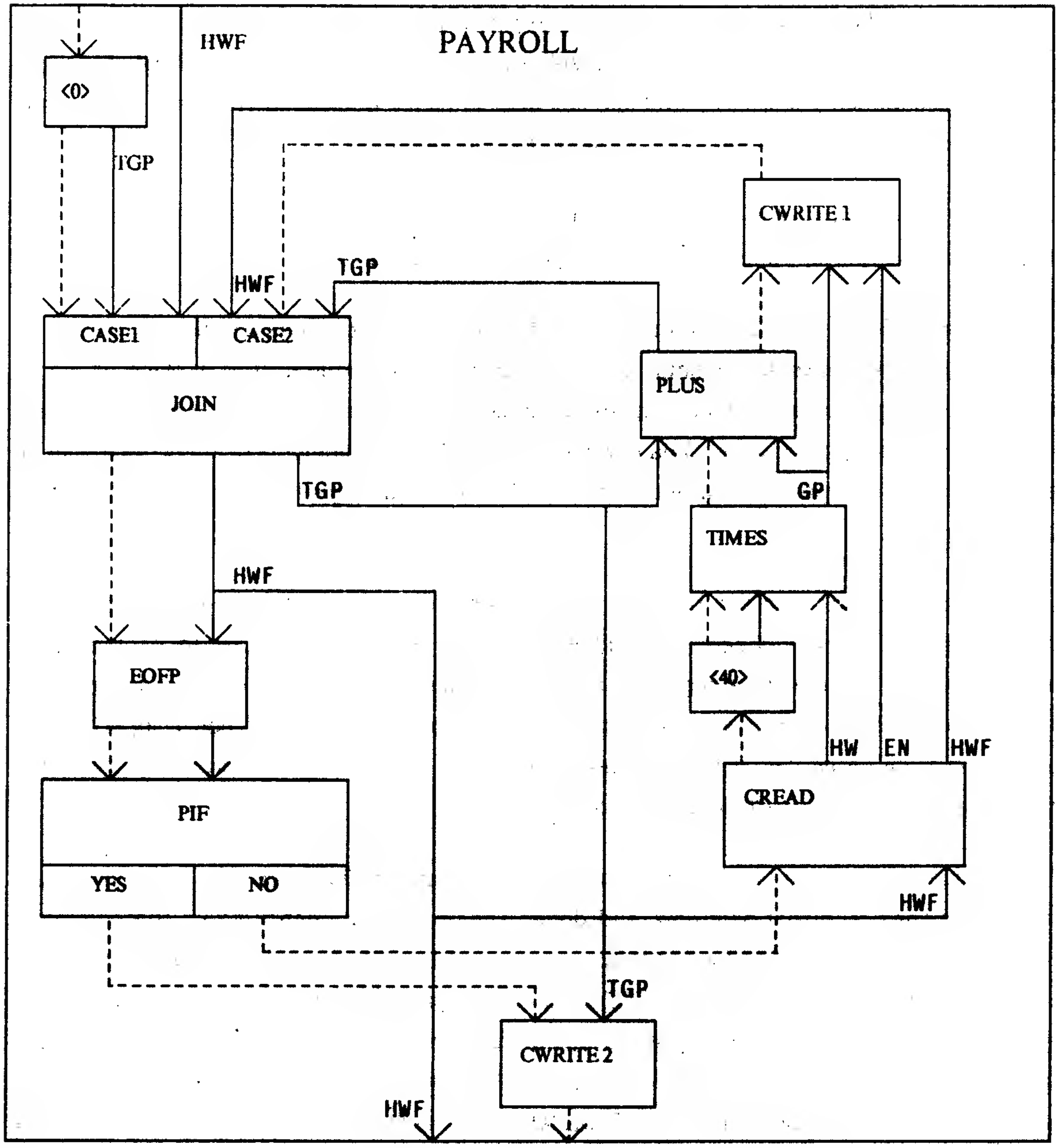
The segments of a surface plan are connected via control flow links and data flow links. A data flow link is a link between two data "ports". A port can be thought of as a place where an incoming or outgoing data value can be stored. Each segment has associated with it a unique port for each input and output data value. Data flow between any two subsegments of the surface plan, or between a port of the supersegment and a port of one of its subsegments, is represented by an explicit data flow link.

A control flow link is a link between two segment "cases". Each case corresponds to a particular control environment. Each segment has associated with it a unique case for each possible control flow path into and out of the segment. A case for incoming control flow is called an "in-case" and a case for outgoing control flow is called an "out-case". For example, a split has a single in-case, and at least two out-cases. As with data flow, control flow information is known only within the supersegment of the two segments involved in the flow.

Consider the simplified surface plan for PAYROLL shown in Figure 25. The boxes represent segments, solid lines represent data flow, and dotted lines represent control flow. The outermost large box represents the segment for PAYROLL itself. (This example has been simplified in several ways. First, the computation of EMPLOYEE-COUNT has been entirely eliminated. Second, for brevity, the file open and file close functions have been removed. Third, the data flow for all of the file objects except for the flow associated with the HOURLY-WAGE FILE-OBJECT (HWF) has been removed. The HOURLY-WAGE FILE-OBJECT was left in so that the operation of the EOFP predicate could be understood.)

Several of the features of surface plans can be seen in this example. First of all, note the control flow throughout the plan. There is a large control flow loop that encompasses most of the program; namely the main read/write loop. Control remains within this loop as long as control passes through the NO case of the PIF, which in turn occurs for as long as EOFP yields a FALSE boolean.

Fig. 25. Partial Surface Plan for PAYROLL



HWF	=> HOURLY-WAGE FILE-OBJECT	TGP	=> TOTAL-GROSS-PAY
HW	=> HOURLY-WAGE	GP	=> GROSS-PAY
EN	=> EMPLOYEE-NUMBER		

This process is initiated when control is passed to CASE1 of the JOIN and terminates the first time control passes to the YES case of the PIF.

Now examine the data flow. In particular, note the flow associated with TOTAL-GROSS-PAY (TGP) or HOURLY-WAGE-FILE (HWF). The initial value for the flow is passed through CASE1 of the JOIN into the main loop. Subsequent values are passed around the loop through CASE2 of the JOIN. This looping of the data continues, with each new value for the flow depending on its previous value, until the loop terminates in which case the final value is received outside the loop.

Given that the plan shown in Figure 25 needed to be simplified from the actual surface plan for PAYROLL (a relatively trivial program) in order to make it at all comprehensible to the human eye, it should be obvious that the surface plan contains large quantities of relatively mundane and unorganized information. It would be a very difficult and expensive task to try to match portions of the surface plan with any patterns that might represent fairly global features of the program. What is needed is more organization of the available information.

4.2 Analyzed Plans and Plan Building Methods

A surface plan can be analyzed in terms of plan building methods (PBMs). The PBMs are a set of stereotyped ways in which plan segments can be aggregated into canonical groupings. An instance of a PBM corresponds to a logical locality in the program, not necessarily a locality in the actual code for the program. Each PBM has a unique set of "roles" associated with it. A segment created to represent an instance of a PBM has a set of subsegments each of which fills one of the roles of the PBM. Each subsegment can only fill one role of one PBM. Therefore, each segment in the analyzed plan will have exactly one immediate superior except for the single most superior segment.

The analysis process begins by searching for a set of subsegments of the surface plan that can be grouped together according to the restrictions of one of the PBMs. A new segment is created to represent the grouping. All of the data flow and control flow information pertaining to any of the subsegments is included in the description of the new segment, and the description of the original supersegment is appropriately modified. The search process is then repeated with the newly created segment considered to be indivisible. The process continues until a grouping subsumes the entire plan. The result is a hierarchical structure in which each segment, except those corresponding to primitive functions, is an instance of one of the PBMs. The leaves of this hierarchy are the same primitive segments that comprised the surface plan for the program.

Fig. 26. Taxonomy of Plan Building Methods

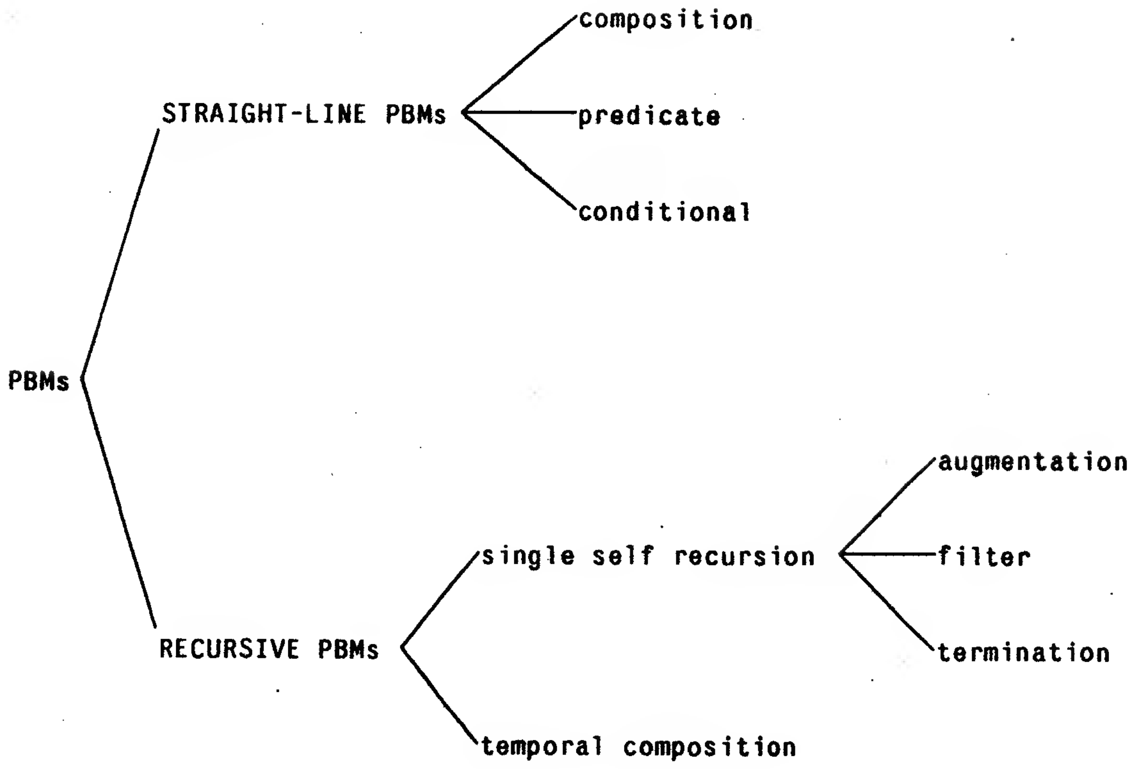


Figure 26 gives a taxonomy of PBMs. As can be seen in the figure, PBMs can be broken into two major categories: "straight-line" and "recursive". This distinction is based upon the manner in which the segments that instantiate the PBMs interact with control flow. The recursive PBMs are used to express the portions of programs that involve loops of control flow while the straight-line PBMs are used to express the portions of programs that do not involve loops.

4.2.1 Straight-line PBMs

There are three straight-line PBMs: "composition", "predicate", and "conditional". The PBM "composition" allows for the combination of an arbitrary number of simple segments into a single simple segment; no splits or joins can be included. In the general case, the data flow links in a composition will form a collection of (possibly unconnected) directed acyclic graphs. Each of the subsegments of the composition fills an "action" role.

The PBM "predicate" is a generalization of the primitive split. It has a single in-case, but an arbitrary number (at least 2) of out-cases. The control flow links of a predicate will form a directed acyclic graph. The subsegments can be either primitive splits or other predicates, or primitive joins, which fill roles called "pred" and "join", respectively. A join subsegment acts to form the logical OR of the predicates that supply it with control flow. A predicate subsegment that receives control flow from another predicate subsegment forms the logical AND of itself and the predicate that supplies the control flow. By using these constructs in combination, predicates of arbitrary complexity can be built.

The PBM "conditional" is an embodiment of the structured programming concept of a conditional. It has a single "pred" role, filled by a subsegment that is an instantiation of the PBM predicate, that controls which of several "actions" will be executed. The action roles can be filled by any simple segment. In addition, it contains a single "join" role, filled by a join segment, that collects the control flow from all of the actions. An instance of the PBM conditional has a single in-case, and a single out-case; it is a simple segment. A conditional also has the very useful property that during any given execution of the conditional *exactly one* of the actions will be executed. A conditional can have an optional "initialization" role which can be filled by any simple segment. The initialization subsegment is executed before the predicate and therefore is executed regardless of the result of the execution of the predicate.

4.2.2 Recursive PBMs

The recursive PBMs are designed to handle loops and other forms of single self recursion. A program is single self recursive if it contains exactly one recursive call to itself, either directly or indirectly. A loop is an example of single self recursion since it can be expressed in terms of tail recursion. Other forms of recursion cannot currently be analyzed in terms of PBMs. However, since COBOL does not support any type of recursion except loops, the current PBMs are sufficient for the analysis of COBOL programs.

The most general recursive PBM is simply called "single self recursion" (SSR). An SSR has three roles; an optional "initialization", a "body", and a "recurrence". The initialization is a simple segment that is only executed once, while the body is executed repeatedly. The recurrence represents a recursive instance of the body. Therefore, it is placed in the body at the point of the recursive call to the body, and it will have the same ports and cases that the body has.

Since the recurrence subsegment is a recursive instance of the body, and the recursion can potentially occur to infinite depth, inclusion within the recurrence of the entire structure of the body would result in an infinite graph. To allow the graph to remain finite while still capturing the notion of a potentially infinite recurse, the recurrence is given no internal structure, but instead is linked to the body by a special recurrence link. Then, during execution of the body, if the next segment to be executed is the recurrence, the values in the incoming data flow ports of the recurrence are transferred to the corresponding ports of the body and control is passed to the body via the recurrence link and the body is executed again. In this way, control and data flow is circulated around in the execution without the existence of any explicit control or data flow cycles in the plan. This lack of control and data flow cycles is very helpful in certain types of symbolic evaluation in which symbolic values are pushed along control and data flow links.

The drawback of the PBM SSR is that its body can be arbitrarily complex and the recurrence buried arbitrarily deep within it. It would be useful to be able to break single self recursions into smaller, less complex pieces. This is done via the PBM "temporal composition".

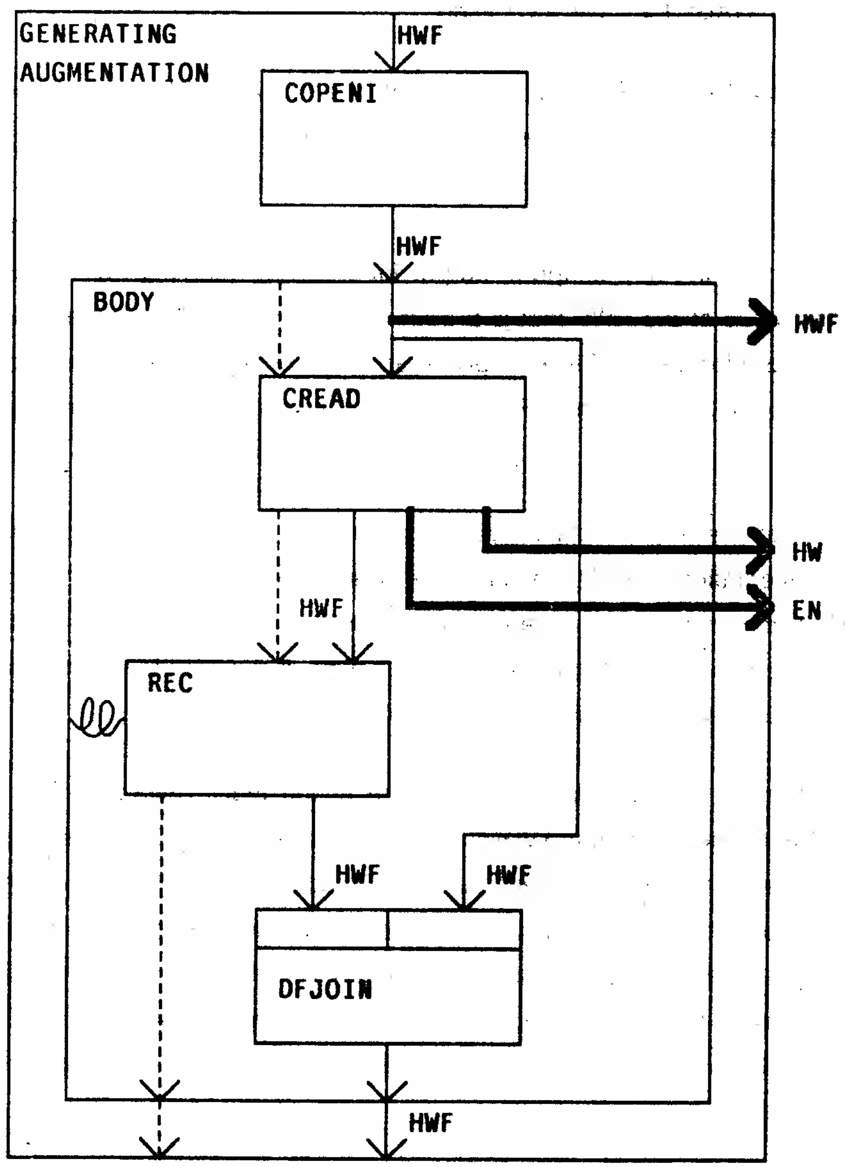
The PBM "temporal composition" is similar to the straight-line PBM composition except that all of its subsegments are instances of the PBM SSR instead of straight-line PBMs. In addition, since some of the subsegments may produce data values recursively that are used in other subsegments, some of the data flows between subsegments represent a temporal sequence of values instead of a single value.

The temporal sequences of values are called "temporal data flows". A temporal data flow into a segment is called a "temporal input", while a temporal data flow out of a segment is called a "temporal output". These temporal flows can be viewed as streams, and the subsegments of a temporal composition that interact with them can be viewed as stream operators. More will be said later about this view of temporal composition and temporal data flows.

Three restricted forms of the PBM SSR, called "augmentation", "filter", and "termination", are used to form meaningful fragments of temporal compositions. In order for an SSR to be an augmentation, the body of the SSR must be constrained in the following ways. First, the body of the augmentation must have a single in-case and a single out-case. Second, the body must have only two subsegments. One of them must be the recurrence. The other, called the "augmentation function", can be any simple segment.

The augmentation function is executed once for each recursive execution of the augmentation. The execution of the augmentation function may use and/or produce data values that are passed into and/or out of the augmentation. These data values are passed via temporal data flows. If the augmentation function only uses data values that are produced within the augmentation, then the augmentation is called a "generating augmentation" or simply a "generator". If the augmentation function uses some data values that are produced outside the augmentation, then the augmentation is called a "consuming augmentation" or simply a "consumer".

Fig. 27. Generating Augmentation in the Analyzed Plan for PAYROLL



Consider the simple example of an augmentation shown in Figure 27. The bold lines represent temporal data flow and the curly line represents the recurrence link. This augmentation is the generator for the temporal composition in PAYROLL. The initialization of the augmentation opens the file HOURLY-WAGE FILE-OBJECT (HWF) for input. The augmentation function is a CREAD acting on the HWF. Temporal outputs are created for each of the outputs of the CREAD function, as well as for the HOURLY-WAGE FILE-OBJECT itself.

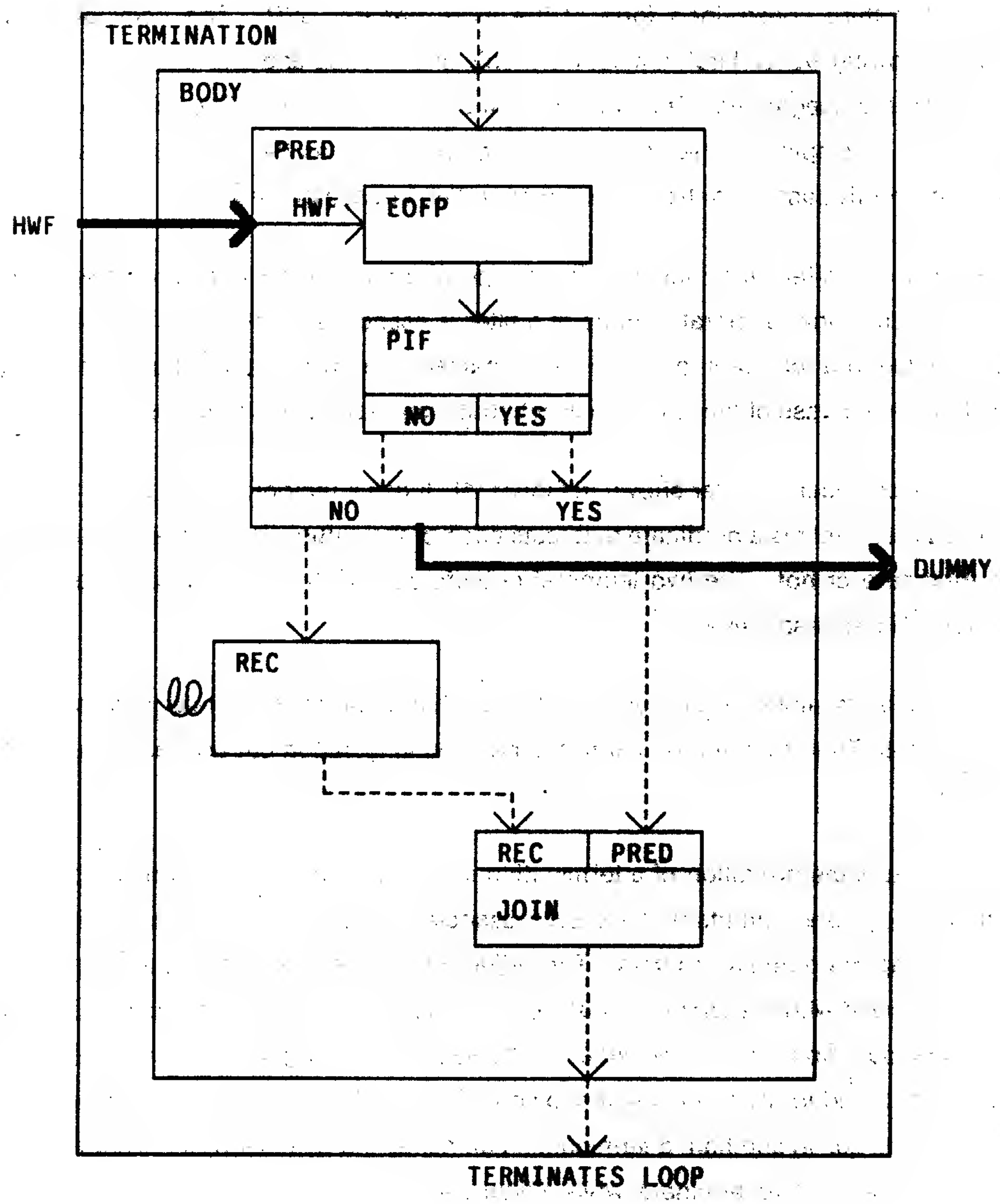
Let us examine the data flow associated with the HOURLY-WAGE FILE-OBJECT in more detail. The HWF is fed into the CREAD the first time from the COPENI initialization. All subsequent values of the HWF used by the CREAD actually come from the output of the CREAD itself through the recurrence segment. In this way, the values for the HWF are fed back in a loop without any loop in the data flow itself. Note that the non-temporal output for the HWF (coming out of the bottom of the augmentation) is the value of the HWF that is produced the last time the augmentation is executed, while the temporal output for the HWF is a temporal sequence of all the values that the HWF data flow assumes at the input to the CREAD. The DFJOIN is not a control flow join but is merely used as a data flow join.

Note that since there is no way for control flow to be passed to the out-case of the body, execution of an augmentation in isolation will never terminate. In addition, a consumer cannot be repeatedly executed in isolation as it needs to receive temporal data flow from outside itself. Therefore, an augmentation cannot stand alone within a plan. It is meant to be a meaningful fragment of a temporal composition, and can only be used as such.

A "termination" is the second restricted form of the more general SSR. The body of a termination is constrained in the following ways. First, as subsegments, it has a recurrence and a split segment. The split segment fills the "pred" role and is additionally called the "termination test". Second, one of the out-cases of the termination test must pass control to the in-case of the recurrence, and at least one of the out-cases of the termination test must pass control to an out-case of the body. An out-case of the body will receive control flow from both an out-case of the recurrence and an out-case of the termination test. This calls for the inclusion of the appropriate number of joins as subsegments of the body.

For example, consider the termination for the temporal composition in PAYROLL shown in Figure 28. The termination function, EOFP, tests the temporal input HOURLY-WAGE FILE-OBJECT (HWF) to determine whether to pass control to the recurrence or to the out-case of the body. Execution of the termination will continue as long as end of file has not yet been reached. As soon as the EOFP predicate senses that end of file has been reached on the HWF, control is passed to the out-case of the body, and the recursive execution of the termination stops. The DUMMY temporal data flow will be explained later.

Fig. 28. Termination in the Analyzed Plan for PAYROLL



A termination is the only fragment of a temporal composition that can terminate by itself, and is the only fragment that can cause the temporal composition as a whole to terminate. Therefore, it is the only fragment that passes control flow to an out-case of the temporal composition. Nonetheless, since it requires temporal input, it cannot stand alone, and is only used as a fragment of a temporal composition.

A filter is the third constrained form of the more general SSR. The body of a filter is constrained in the following ways. First, it must have only one in-case and one out-case. Second, it must have exactly three subsegments. One of them must be the recurrence. The other two segments are a split and a join. The split has the same number of out-cases as the join has in-cases. Each out-case of the split directly passes control to a corresponding in-case of the join.

The filter has the effect that some of the temporal inputs to the filter are broken up into temporal outputs. Each of the temporal outputs of a filter is associated with one of the out-cases of the split. A given value in a temporal input will be contained in a corresponding temporal output if control is passed to the out-case of the split with which that temporal output is associated.

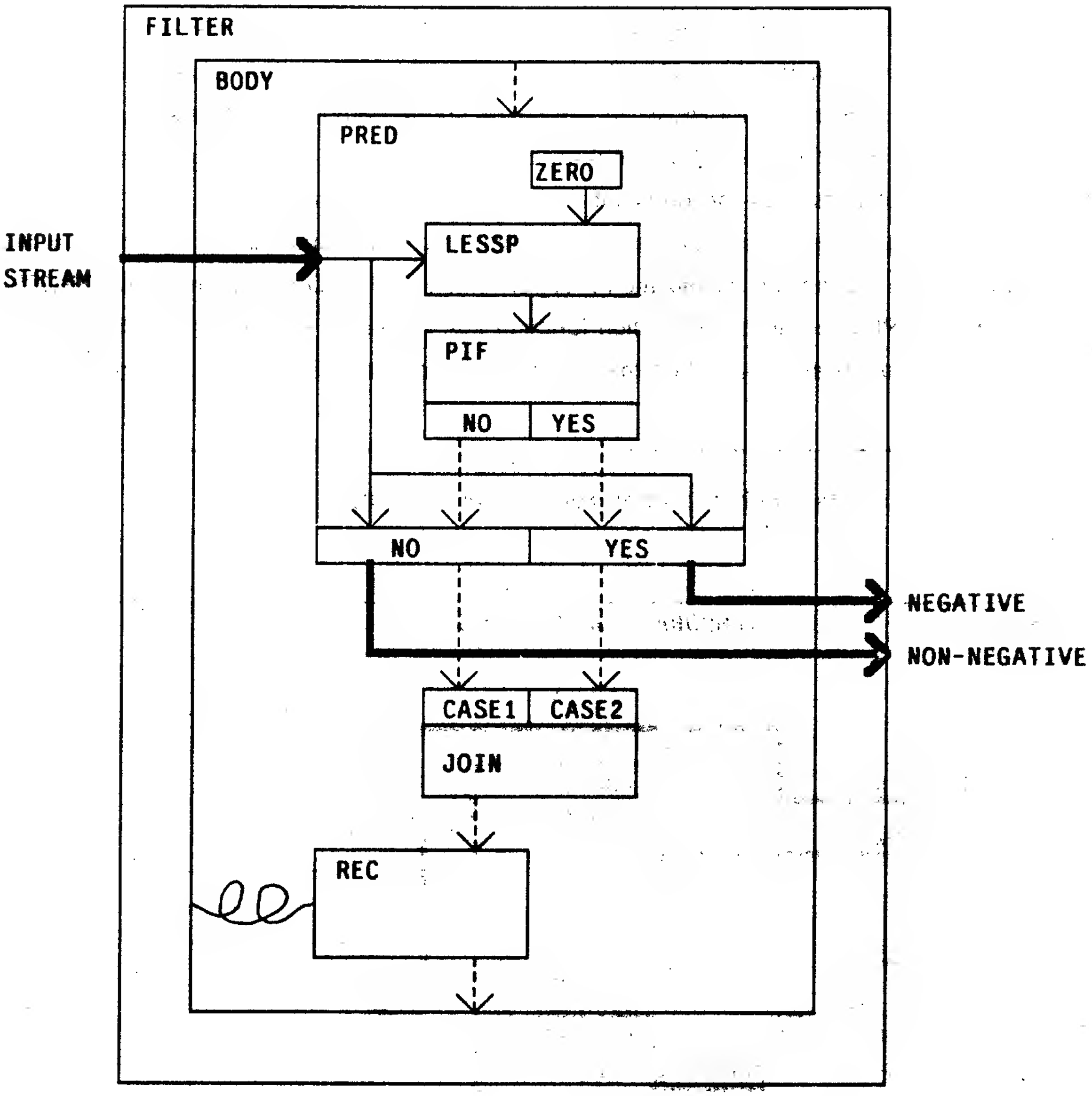
For example, consider the filter shown in Figure 29. It has a temporal input which is a stream of numbers. The split segment is a predicate that tests each of the values in the temporal input to see if they are less than zero or not. The two temporal outputs contain the negative and non-negative values in the temporal input respectively.

Note that a filter is similar to an augmentation in that it cannot terminate in isolation and it requires temporal input. Therefore, a filter cannot to be used in isolation but only as a fragment of a temporal composition.

In many respects the operation of a termination is very similar to the operation of a filter. A termination can have temporal outputs that are each associated with an out-case of the termination test, similar to the temporal outputs of filters. The difference lies in the fact that a filter will *select* certain values interspersed within a temporal data flow, while a termination will *truncate* values off of the end of a temporal data flow. This difference can be seen by viewing the difference between the DUMMY data flow produced in the termination shown in Figure 28 and the two temporal outputs of the filter shown in Figure 29. In addition, a termination can cause the execution of the entire temporal composition in which it appears to terminate, while a filter cannot.

Although it is easier to understand the internal structure of the fragments of a temporal composition in terms of their function during execution, it is often easier to describe the contribution of each subsegment of a temporal composition to the entire operation of that temporal composition by viewing the subsegments as stream operators. In this way, the external properties of the

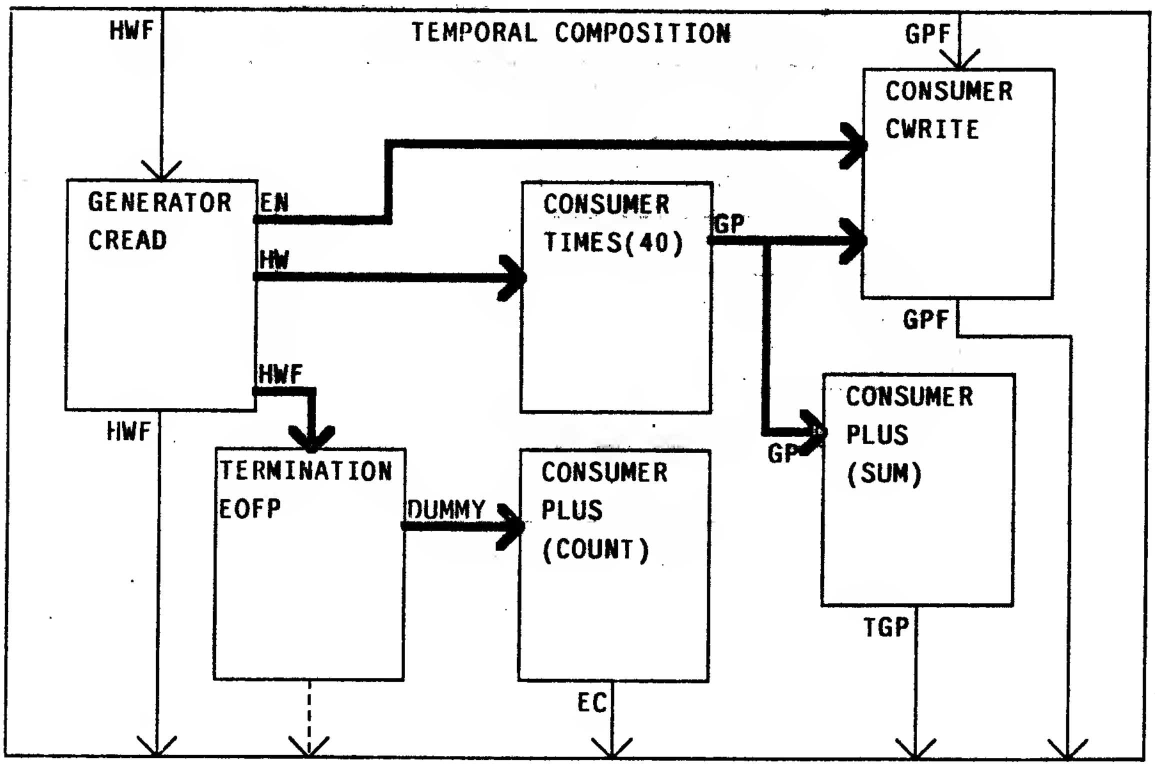
Fig. 29. Example Filter



subsegments can be expressed in a much more succinct manner. In addition, the function that the temporal composition represents can often be described without considering the values of certain of the input data values; information without which execution is not possible. This allows for a description of the *general* function represented by a particular temporal composition.

For example, consider the temporal composition in the analyzed plan for PAYROLL shown in Figure 30. A detailed view of the analyzed plan for the first two subsegments, the generating augmentation and the termination, have already been given above. The other augmentations have internal structures very similar to the CREAD augmentation and therefore will not be shown in detail. The first of these takes the HOURLY-WAGE (HW) temporal output of the CREAD augmentation as its temporal input and multiplies it by the constant 40, producing a temporal output for GROSS-PAY (GP). This temporal output is, in turn, passed to two additional augmentations. One of them is the CWRITE augmentation that has an initialization that performs a COPENO operation on the GROSS-PAY FILE-OBJECT (GPF), and an augmentation function that writes the values of GROSS-PAY into that file. The other one has an initialization that produces the constant ZERO and an augmentation

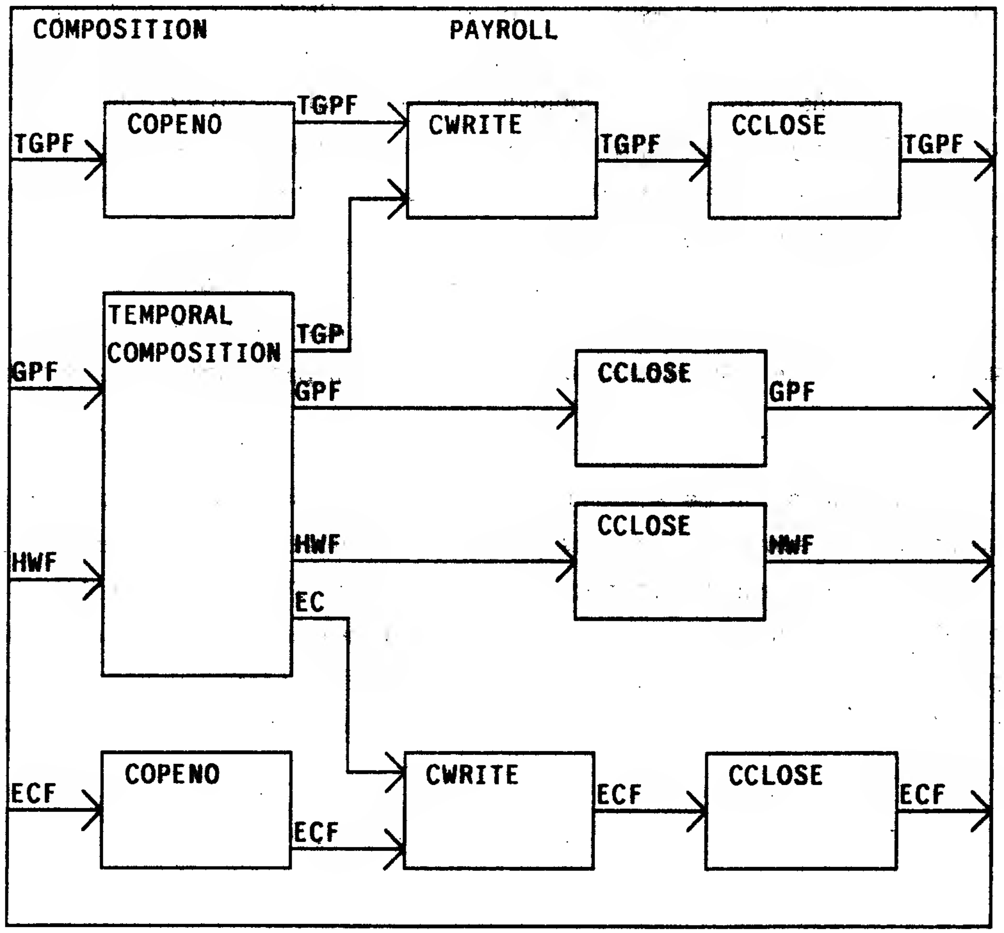
Fig. 30. Temporal Composition in the Analyzed Plan for PAYROLL



function of PLUS that computes the sum of GROSS-PAY. The non-temporal output of this augmentation is TOTAL-GROSS-PAY (TGP) and is passed to an output port of the temporal composition to be written outside of the temporal composition.

The remaining augmentation performs the computation for EMPLOYEE-COUNT (EC) and also has an initialization that produces ZERO and an augmentation function of PLUS. The difference is that the second argument to the PLUS is the constant ONE. Therefore, all data values that are needed by the augmentation are Internally generated. The function of the DUMMY temporal data flow, generated in the termination and associated with the NO case of the EOFP predicate, is to provide a control signal to the consuming augmentation which tells it how many times to execute. The non-temporal output EC is passed out of the temporal composition.

Fig. 31. Analyzed Plan for PAYROLL



4.2.3 Analyzed Plan for PAYROLL

Now that most of the components of the analyzed plan for PAYROLL have been described, we can take a look at the entire plan shown in Figure 31. The top level segment of the analyzed plan is a composition in which most of the subsegments perform primitive file operations. The only exception is the central temporal composition, the internal structure of which has already been given.

This example should make it clear that although the entire hierarchy of the analyzed plan for a program can be quite complex, any particular level in the hierarchy is fairly simple. It is the hierarchical nature of an analyzed plan, as well as the simplicity at each level in the hierarchy, that contribute to the fact that an analyzed plan is a much more organized source of information about a program than either the original code for the program or the surface plan for the program.

4.3 Conclusion

By comparing the COBOL code for PAYROLL with the analyzed plan for that program, it can be seen that the analyzed plan is much easier to reason about. The PBMs group information that may be contained in distant parts of the actual code into neat functional localities. This locality makes it possible to make conclusions about certain computations without considering the entire program. In addition, a programming language like COBOL has many constructs for the transfer of data values from one place in the program to another. The analyzed plan for the same computation uses data flow as the single construct for data value transfer. The stereotypicality of the analyzed plan further reduces the number of distinct possibilities that need to be considered at any one step in a deductive process. It is the reduction in the number of facts about the program which need to be considered simultaneously that makes the PBM representation of a program particularly useful for abstract processing.

A given computation can be subdivided into smaller chunks in several ways including processes, subprograms, streams, and data abstractions. The analysis described here, via the PBM temporal composition, uses the streams abstraction. This is critical to the translation of the COBOL programs into HIBOL. Since HIBOL is essentially a method of expressing data processing functions in terms of operations on streams (data sets), the initial analysis of the COBOL programs in terms of stream operators (augmentations, terminations, and filters) is a significant first step in the translation of COBOL into HIBOL.

5. Current Implementation of the Translation Process

The three formalisms for the description of data processing programs discussed so far, COBOL, HIBOL, and analyzed plans, are the result of the work of others. This and the following two chapters describe the research effort of this thesis.

5.1 General Description

The diagram shown in Figure 32 highlights the current implementation of the SATCH system. Starting with a COBOL program, the COBOL parser (implemented by Burke) produces two distinct outputs. First, information is extracted from the DATA DIVISION and placed in a file to be used later in the data division query phase of the translation process. Second, the PROCEDURE DIVISION is transformed into a lisp-like format that represents the computation in terms of the primitive functions described in Appendix I. This representation of the PROCEDURE DIVISION is then used by the plan extraction and analysis phase (implemented by Waters) to produce the analyzed plan as described in the previous chapter.

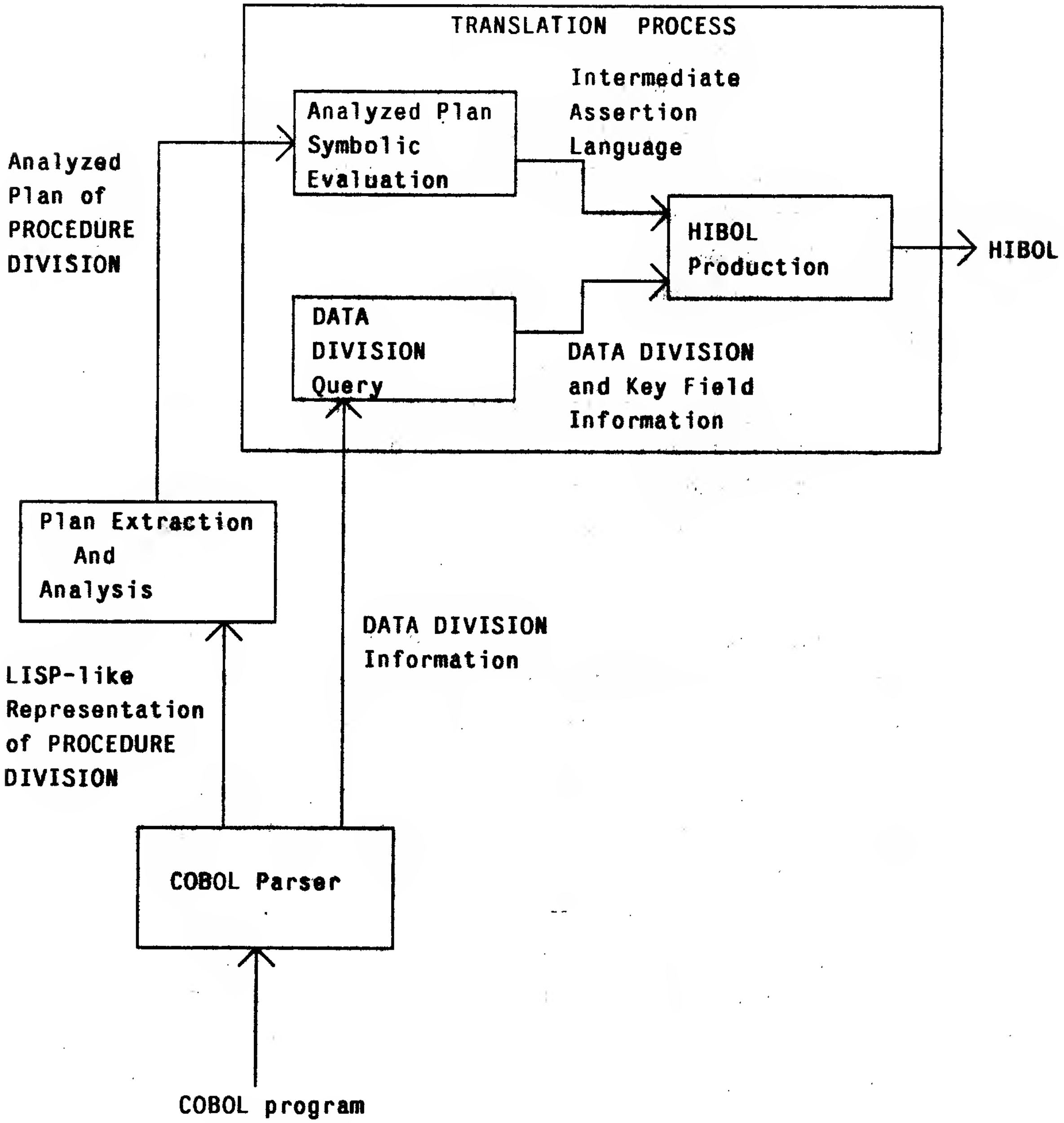
The translation process is divided into three subprocesses. The first two subprocesses, the symbolic evaluation of the analyzed plan and the data division query, can theoretically proceed in either order. For reasons that will become clear, the symbolic evaluation of the analyzed plan is done before the data division query. Since the third subprocess, HIBOL production, uses the results of the first two subprocesses, it cannot proceed until they are completed.

The symbolic evaluation of the analyzed plan is by far the most time consuming of the three subprocesses. It proceeds by making an assertion about the value of every output data port on every segment, and an assertion about every out-case of every split segment.

A key feature of COBOL programs is that they do not return values. Therefore, the only way they can produce results is by the side effect of writing data values into files. This means that the only information that needs to be transferred from the symbolic evaluation of the analyzed plan for the program to the HIBOL production phase are the values of the data flows that are used as arguments to CWRITE and CREWRITE. After this information is gathered from the analyzed plan, the plan is no longer needed in the translation process. The syntax and semantics of the intermediate language that is used for assertions and to transfer information to the HIBOL production phase will be discussed later in some detail.

Much of the information that is originally contained in the DATA DIVISION of the COBOL program is transferred to the translation phase directly from the COBOL parser and is not passed to the plan analysis phase at all. Unfortunately, some specific information that is needed in order to

Fig. 32. Current SATCH Implementation


TRANSLATION PROCESS
Analyzed Plan Symbolic Evaluation
Intermediate Assertion Language
HIBOL Production
HIBOL
DATA DIVISION Query
DATA DIVISION and Key Field Information
Analyzed Plan of PROCEDURE DIVISION
Plan Extraction And Analysis
DATA DIVISION Information
LISP-like Representation of PROCEDURE DIVISION
COBOL Parser
COBOL program

produce the HIBOL is not directly contained in the the DATA DIVISION of the COBOL program, nor can it be gleaned from the analyzed plan for the PROCEDURE DIVISION. In particular, in most instances it is impossible to tell which fields in a data file represent key fields and which are data fields. This information is gathered in the data division query subprocess.

The subprocess that produces the actual HIBOL uses the information gathered in the previous two subprocesses. In doing so, it makes certain assumptions about the form of the original COBOL program. These assumptions will be discussed in the next section. It also uses extensive knowledge about the semantics of HIBOL in an attempt to produce HIBOL that is a faithful translation of the semantics of the original COBOL program without redundantly specifying restrictions that are implicit in HIBOL. Elimination of the specification of implicit restrictions leads to the production of HIBOL code that might be harder for a HIBOL parser to process, but that is easier for a human reader to understand.

5.2 Range of COBOL Programs Currently Translatable

The current implementation of the translation process makes use of certain assumptions about the type of COBOL program that is represented in the analyzed plan. Some of these assumptions stem from the limits of the expressibility of HIBOL. Others stem from a desire to reduce the domain to a manageable size.

The translation process is designed to work on three basic types of programs. The simplest type of program is one which reads in a file and outputs another file. The input and output files must have the same key fields. In addition, the output file contains exactly one record for each record in the input file and each record in the output file has the same values for the key fields as the record in the input file that was used to create it. In HIBOL terms, this means that the actual key space of each output file is identical to the actual key space of the input file. The PAYROLL and DBINIT programs shown in Section 1.4 are examples of this type of program.

The second type of program is an extension of the first in which the computation of the value of the data fields in the output file requires information contained in the data fields of two (or more) input files. The input files and the output file must all have the same key fields. Since the computation for a data field in the output file requires information from a record in each of the input files, the output file only contains a record if a record with identical key field values appears in all of the input files. In HIBOL terms, this means that the actual key space of the output file is the intersection of the actual key spaces of the input files. The PAYROLL2 program shown in Section 1.4 is an example of this type of program.

The third type of program also produces an output file using information contained in two (or more) input files but *each record* in the output file only uses information contained in a record from *one* of the input files. This occurs, for example, when a program performs a file merge operation. The input files and the output file must have the same key fields. Since a value for a data field in the output file can be computed from information contained in a record in any one of the input files, the output file contains a record for each unique set of key field values appearing in any of the input files. A record in the output file is given the same values for the key fields as the record in one of the input files from which it was created. In HIBOL terms, this means that the actual key space of the output file is the union of the actual key spaces of the input files. The LOC-LIST program shown in Section 1.4 is an example of this type of program.

All of these types of programs have certain features in common. First, all top-level loops in the COBOL program are logically driven by file reads and terminated by end of file predicates. Since HIBOL has no explicit loop construct, loops other than these cannot in general be reasonably expressed in HIBOL. Second, these types of programs do not contain non-local error exits from any of the loops or from the program itself. Such non-local jumps are usually not expressible in HIBOL and also are not well expressed within analyzed plans. Third, all input data files (and therefore all output data files) are homogeneous. That is, all records in a file are assumed to have data and key fields which contain the same type of information as the corresponding data and key fields in all the rest of the records in that file. This means that the file cannot contain any singular header or trailer records with a different interpretation from the rest of the records. It might be possible to produce HIBOL from COBOL programs that do access files that contain header or trailer records, but the translation process would have to generate data-sets for these records that were independent from the data-sets generated for the rest of the records. Currently, a single data-set is generated for each data field in each data file.

Certain additional restrictions are also required. First, it is assumed that all input files are read sequentially, and all output files are written sequentially. In a later chapter suggestions are made as to how this constraint might be eliminated as long as the program still falls into one of the three basic categories. Note that it follows from this constraint that, in programs of the second and third type (intersection and union), the input data files used in conjunction to produce the output file must be sorted in the same key field order.

Second, it is assumed that the COBOL program contains no nested loops. This is a rather harsh constraint and would have to be eliminated before translation from COBOL into HIBOL could be applicable to the real world. One group of programs eliminated by this assumption are those that produce subtotals for certain data fields in a record as a secondary key field changes value.

Third, it is assumed that no output is performed on any files other than data files. That is, it is assumed that the program produces no formatted output reports. Although HIBOL does have a report generation feature, the generation of formatted reports is an orthogonal issue to the rest of the semantics of HIBOL. In a later chapter, some suggestions are made as to how translation of formatted reports might proceed.

Fourth, for simplicity, it is currently assumed that, within a given program, all key field names from different files that actually correspond to the same key are identical. This constraint is particularly easy to eliminate, and a method for doing so will be discussed in a later chapter.

The above constraints are not as restrictive as they might seem. The three basic program structures discussed above represent the heart of the domain of programs that can be expressed in HIBOL. In addition, programs which incorporate other features that do not interfere with the main read loops can still be translated. For example, a single program can produce output files from input files using any or all of the three basic strategies, so long as the read/write loops used to produce those output files are completely separate from one another and therefore cannot interact. Also, reduction operations that produce grand totals are allowed because they do not require nested loops. It is also possible to translate programs which do not produce a record in the output file for each set of key values that could cause a record to be produced. For example, In the LOC-LIST program, a output record is not produced for a record in the transaction file if that record contains a "D" in the delete-flag field. (Note that it is not permissible to *add records* to the output file in a similar fashion).

Unfortunately, the current implementation of the translation process does not verify that the program that it is processing adheres to the assumptions and/or restrictions discussed above. A more robust system would have to do significant checking to determine if the program that it is processing fell within the domain of programs that it was designed to translate.

5.3 Brief Example

Before delving into all of the detail of the current implementation process, let us examine its operation on a simple example program, namely PAYROLL. For simplicity, only the processing needed to produce the HIBOL for the output data-set GROSS-PAY will be discussed; TOTAL-GROSS-PAY and EMPLOYEE-COUNT will not be considered. This discussion is not meant to make the operation of the translation process crystal clear, but merely to give a flavor for the type of processing that is taking place.

5.3.1 Symbolic Evaluation of the Analyzed Plan

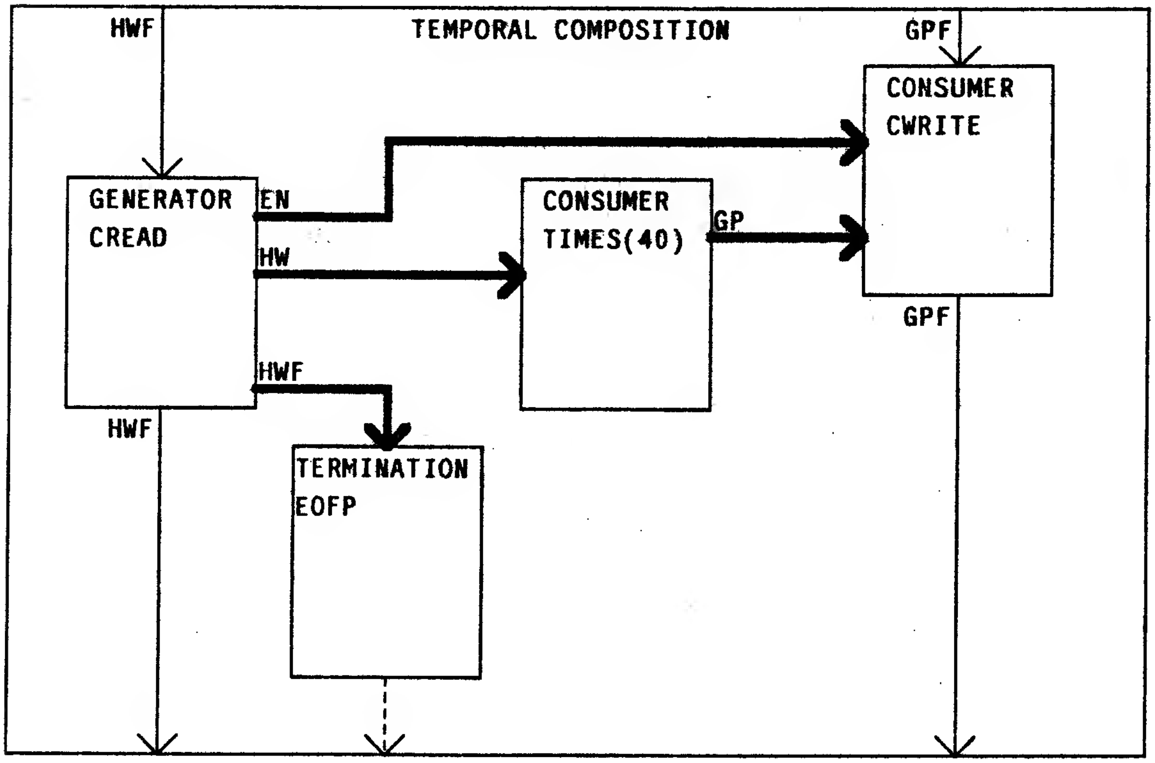
Let us consider the portion of the analyzed plan for PAYROLL shown in Figure 33. This figure shows an abbreviated version of the analyzed plan for the main temporal composition (and is, in fact, a subset of Figure 30). The subsegments are symbolically evaluated in an order that is consistent with their control and data flow dependencies (left to right in Figure 33).

The first subsegment of the temporal composition to be symbolically evaluated is the generating augmentation, which has CREAD as its central function. The assertion that is formed for the HOURLY-WAGE (HW) output port of that subsegment is:

```
(CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
```

This assertion specifies that the value at this output port is the result of reading the HOURLY-WAGE data field in the HOURLY-WAGE-IN file.

Fig. 33. Partial Analyzed Plan for PAYROLL



Also pertinent to this discussion is the assertion that is formed for the HOURLY-WAGE-FILE-OBJECT (HWF) output port of this augmentation:

```
(SEFO HOURLY-WAGE-IN)
```

This assertion specifies that the value at this output port is the file-object HOURLY-WAGE-IN that has been side-effected by the read operation (SEFO is an acronym for "Side Effected File Object").

The next subsegment to be evaluated is the termination subsegment, which has EOFP as its central function. The assertion that is formed for the single out-case of the termination is:

```
(EOFP (SEFO HOURLY-WAGE-IN))
```

This assertion specifies that the termination subsegment (and, therefore, the temporal composition as a whole) terminates when end-of-file has been reached on the HOURLY-WAGE-IN file.

The next subsegment to be evaluated is the consuming augmentation that has TIMES as its central function. This augmentation has the effect that the value for the incoming data flow is multiplied by forty. Accordingly, the assertion that is formed for the GROSS-PAY (GP) output port of that subsegment is:

```
(TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE) 40.)
```

The last subsegment to be symbolically evaluated is the consuming augmentation that has CWRITE as its central function. When this augmentation is evaluated, a record is made of the fact that the output data field GROSS-PAY is associated with the assertion shown above. In addition, the predicate which controls how often it is written (the predicate assertion taken from the out-case of the termination) is stored.

5.3.2 DATA DIVISION Query

In the DATA DIVISION Query phase, the user of the SATCH system is asked to supply the key fields for each of the files appearing in the COBOL program. In this example, the user specifies that EMPLOYEE-NUMBER is the key field for both the HOURLY-WAGE-IN file and the GROSS-PAY-OUT file.

5.3.3 HIBOL Production

In the HIBOL production phase, a new assertion for GROSS-PAY is formed by combining the old assertion for GROSS-PAY with the predicate which specifies under what circumstances it is written. Since GROSS-PAY is written within a temporal composition, the predicate that is used is the negation of the predicate which terminated that temporal composition (stored during the symbolic evaluation phase). Therefore, the predicate that is used is:

```
(NOT (EOFP (SEFO HOURLY-WAGE-IN)))
```

The new assertion (the form of which is not important here) specifies that the value of

```
(TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE) 40.)
```

is written into the output data field GROSS-PAY for every value of the input data field HOURLY-WAGE that appears in the HOURLY-WAGE-IN file.

This assertion is then translated into the corresponding HIBOL statement:

```
GROSS-PAY IS (HOURLY-WAGE * 40.)
```

5.4 Symbolic Evaluation of an Analyzed Plan

As stated above, the symbolic evaluation of an analyzed plan for the PROCEDURE DIVISION of a COBOL program proceeds by making assertions about each output port for each segment. The form of an assertion depends on the PBM that was used to form the segment. In some cases, more specific patterns are used to make special case assertions. This is particularly true for augmentation segments.

In addition to assertions for output ports, an assertion is made for each out-case of every split segment. The assertions specify under what condition control will be passed to that case. These assertions differ from the assertions for data ports in that they take the form of predicates instead of object descriptions. That is, they are expressions that use boolean operators instead of the arithmetic and other special form operators that are used to describe objects.

When a given segment is symbolically evaluated, first its subsegments are symbolically evaluated in an order consistent their control flow and data flow dependencies, starting with a subsegment which depends on none of the other subsegments for either control or data flow. Then, after the symbolic evaluation of the subsegments is completed, an assertion is made about each of the segment's output ports, or if it is a split segment, each of its out-cases.

Both predicate and object assertions are made in terms of primitive objects. Therefore, primitive objects will be discussed in the next section. The two sections after that will discuss predicate and object assertions, respectively.

5.4.1 Primitive Objects

The only explicit inputs to a COBOL program are file objects. However, there are also implicit inputs to the program; namely the data and key fields in the files. These two types of input produce two of the three types of primitive objects. The syntax (literals are in bold face and non-terminal symbols are in italics while primitive function names are in the normal font) for file objects is:

```
(SEFO file-name)
```

where SEFO is an acronym for Side Effected File Object. The syntax for the primitive objects resulting from CREAD operations is:

```
(CREAD-VAL file-name field-name)
```

Since several files may have fields that have the same name, the *field-names* that are used are actually buffer-structure path names that uniquely identify a particular field.

The third type of primitive object is a constant. These fall into two subtypes, numeric and literal. The syntax for numeric constants is simply the numeral itself. The syntax for literal constants is:

```
(STRING some-sequence-of-characters)
```

In addition, there is a special constant, **UNDEFINED**, which is the initial value given to every data area that is not explicitly initialized in the DATA DIVISION of the COBOL program.

5.4.2 Predicate Assertions

The simplest instance of the PBM predicate will have two subsegments: an initialization that is one of the primitive boolean functions, for example EQUAL, combined with a PIF. The assertion that specifies the value for the output data port of the initialization is built in an obvious manner. The primitive boolean function is simply combined in prefix order with the values for the objects that it uses as arguments. For example, the predicate assertion that would be formed from a primitive EQUAL function acting on two primitive **CREAD-VAL** objects would be:

```
(EQUAL (CREAD-VAL file1 field-name1) (CREAD-VAL file2 field-name2))
```

Note that *file1* and *file2* might be the same if the fields to be compared are both from the same file.

The output data port of the initialization will be linked via data flow to the input port of the PIF. (Recall that PIF is the split primitive that tests a simple boolean operand). The PIF will have two out-cases. An assertion will be made about one of the cases, called the YES case, that is the same as the assertion that was made about the output data port of the initialization. An assertion will be made about the other case, called the NO case, that is the negation of the assertion made for the YES case. Using the above example, these assertions state that control will be passed to the YES case of the PIF when

```
(EQUAL (CREAD-VAL file1 field-name1) (CREAD-VAL file2 field-name2))
```

is true, and control will be passed to the NO case when

```
(NOT (EQUAL (CREAD-VAL file1 field-name1) (CREAD-VAL file2 field-name2)))
```

is true.

If the primitive split were a PIFNULL instead of a PIF, then the assertions associated with the YES and NO cases would simply be reversed.

The simple predicate described above will only have two out-cases, each corresponding to one of the out-cases of its PIF subsegment. Symbolic evaluation of the predicate is completed by simply making an assertion about each of the out-cases that is identical to the assertion that was made about the corresponding out-case of the PIF.

As stated in Chapter 4, compound predicates are built out of simpler predicates in two ways. One way is for a predicate, call it PRED2, to receive control flow from an out-case of another predicate, call it CASE1 of PRED1. Because of the order in which segments are symbolically evaluated, PRED1 will always be evaluated before PRED2. PRED2 is then evaluated as usual except that the normal assertions that would have been made had it occurred in isolation are each ANDed with the assertion governing CASE1 of PRED1. For example, suppose that PRED2 is the simple EQUAL predicate discussed above, and CASE1 of PRED1 was asserted to be active when some arbitrary predicate, call it *pred1*, is true. Then assertions will be made stating that control will be passed to one case of PRED2 when

```
(AND pred1 (EQUAL (CREAD-VAL file1 field-name1)
                  (CREAD-VAL file2 field-name2)))
```

is true, and to the other case when

```
(NOT (AND pred1 (EQUAL (CREAD-VAL file1 field-name1)
                       (CREAD-VAL file2 field-name2))))
```

is true.

The other way in which compound predicates are formed is when a join segment receives control flow from an out-case of two different predicates. In this event, an assertion is made about the single out-case of the join that is the OR of the two assertions that govern the two in-cases of the join, which in turn are governed by the two predicates that pass control to the join. For example, suppose that the join is passed control from an out-case of two predicates, and that those cases are asserted to be active when two predicates, call them *pred1* and *pred2*, are true. Then an assertion will be made about the out-case of the join that is of the form:

```
(OR pred1 pred2)
```

A compound predicate, in general, can have many joins and simpler predicates as subsegments. It can also have many out-cases, each of which is passed control from an out-case of one of its subsegments. When a compound predicate is symbolically evaluated, first all of the subsegments are evaluated, and then an assertion is made about each of its out-cases which is identical to the assertion that was made about the out-case of the subsegment that passes control to it.

It should be clear that the expressions for the assertions in a compound predicate can be very complex. If the assertions for compound predicates were made according to the rules that have been given so far, they would be in an unreadable form. This is also true of the assertions that are made about complex objects. To reduce this problem, several simplification techniques have been used. These will be discussed in a later section.

5.4.3 Object Assertions

The assertions formed for primitive objects were discussed above. The following three sections will discuss assertions made in primitive segments, segments that are instances of straight-line PBMs, and segments that are instances of recursive PBMs, respectively.

5.4.3.1 Primitive Function Assertions

Assertions about output ports of primitive arithmetic functions that do not fall into any special category are formed in an obvious manner. The primitive function of the segment is combined, in prefix order, with the assertions about the input ports to the segments. For example, suppose that there is a primitive TIMES function that has two input ports. By following data flow links to each of the input ports back to the previous segment, an assertion can be found for each of the input objects. Suppose that the assertions found in this manner are *obj1* and *obj2*. Then the assertion that will be formed for the output port of the segment is:

```
(TIMES obj1 obj2)
```

Special assertions are formed for the primitive functions that perform simple operations on file objects: COPENI, COPENO, COPENIO, CCLOSE, and NTERPRI. All of these functions have the property that they take a file object as their only input and produce a file object as their only output. In all cases, the assertion that is formed for the output is just the primitive object:

```
(SEFO file-name)
```

In addition, special assertions are formed for CREAD functions. The CREAD function is unique in that it takes a single input, a file object, and produces many outputs. One of the outputs is the file object, and the others are all field values. The output assertion for the file object is as above. An assertion is made for each of the other output ports that is of the form:

```
(CREAD-VAL file-name field-name)
```

where the `field-name` is one of the names in the buffer-structure associated with the file in the DATA DIVISION of the COBOL program. These field names are contained within the analyzed plan and do not come from the DATA DIVISION information produced in the COBOL parser.

CWRITE and CREWRITE functions are also handled specially. These functions take a file object as well as a number of other inputs that correspond to fields of that file. The single output is the file object and is given the usual assertion. Symbolic evaluation of these functions also has the side-effect that the assertions that correspond to the fields, along with the associated field names, are placed in a file to be used by the HIBOL production phase of the translation process. The transferal of this Information will be discussed in a later section.

5.4.3.2 Object Assertions Formed in Straight-line PBMs

More complex object assertions are formed within segments that represent straight-line PBMs. The assertions formed within predicate segments have already been discussed. The assertions formed in composition segments and conditional segments are discussed in this section.

In a composition, the assertions that are made about the output ports come from the subsegments that make up the composition. After all the subsegments of the composition have been symbolically evaluated, an assertion is made about each of the output ports that is identical to the assertion associated with the output port of the subsegment that produces data flow to that port. The composition itself lends no special form to the assertions.

A conditional, on the other hand, does cause the formation of a particular type of assertion. Recall that a conditional is composed of a predicate, a group of actions, and a join. The join subsegment not only joins control flow, but also joins data flow. Each output port of the join is associated with as many input ports as the join has in-cases. For example, if the join has three output ports, and four in-cases, then it will also have twelve input ports, three for each of the four in-cases.

Also recall that each of the in-cases of the join is associated with an out-case of the predicate. Because the predicate has the property that exactly one of its out-cases will be active on any given execution of the conditional, the join has the property that exactly one of its in-cases will be active. This causes exactly one of the input ports associated with a particular output port of the join to receive a data value on any given execution.

Since the assertions about the output ports of the join are made during a symbolic evaluation of the conditional, they need to include all the possible values that that output port can assume. This is done by forming a set of predicate-object pairs for each output port. The set for a particular output port is found by associating the predicate that corresponds to each in-case of the join with the data value that the output port would receive had that predicate been true. The syntax of such an object is:

```
(XCASE (pred1 obj1) (pred2 obj2) . . . (predn objn))
```

The keyword XCASE is included as an indicator of the type of object.

Although the syntax of an XCASE construct closely resembles the syntax of a LISP COND construct, the semantics of an XCASE and a COND differ in that the order in which the clauses appear in a COND matters, while in an XCASE the order in which the predicate-object pairs appear does not matter.

5.4.3.3 Object Assertions Formed in Recursive PBMs

The initialization of an augmentation is a simple segment. Therefore, the assertions that are made about its output ports are just those that have been discussed above. The augmentation function is also a simple segment and is also given assertions that are the same as those discussed above. The exception occurs when the augmentation function is a temporal composition. This happens as a result of the nesting of loops. The current implementation of the translation process does not handle this case.

After the initialization and augmentation function have been symbolically evaluated and assertions made about their output ports, assertions are made about the output ports of the augmentation-body. These are made by simply carrying forward the assertions made about the output ports of the augmentation function that correspond to them. Note that this is similar to the way in which assertions are made about the output ports of composition segments.

It is also necessary to make assertions about the temporal and non-temporal outputs of the augmentation. Currently, the temporal outputs are given assertions that are identical to the assertions given to the data ports from which they get their values. Unfortunately, this means that these assertions are indistinguishable from assertions made about non-temporal data flows. The pros and cons of this choice are discussed in another section.

In addition to temporal outputs, an augmentation can also have non-temporal outputs. In the analyzed plans for COBOL programs, these arise in two ways. First, a file object can be side-effected in an augmentation and then passed out of the augmentation to be used in another segment in the plan. In this event, the output port that corresponds to the file object is given the usual file object assertion, namely:

```
(SEFO file-name)
```

The second non-temporal output type results from reduction operators such as COUNT or SUM. The reduction operators are recognized when the augmentation satisfies special criteria. For example, a SUM operation can be recognized when an augmentation has an initialization that produces the constant zero and an augmentation function that is a PLUS. The PLUS function will take one argument from an input to the body that first gets its value from the initialization and subsequently gets its values from the output of the PLUS function. The second argument will be a temporal input to the augmentation. The non-temporal output of the augmentation is then the SUM of the temporal input to the augmentation. The assertion that would be formed in this event is:

```
(SUM obj)
```

where *obj* is the assertion found by following data flow back from the temporal input port to its source.

Terminations and filters are handled in much the same way as augmentations. The subsegments of the filters and terminations that represent straight-line PBMs are evaluated as always and the usual assertions formed. However, the temporal outputs of terminations and filters need to be handled in a special way. These temporal outputs represent stream values, generated in some augmentation, that have been changed by the action of the termination or filter.

Recall that each temporal output of a termination or filter corresponds to a temporal input that has been associated with a certain predicate. In the case of a termination, this predicate indicates at which place the temporal input should be truncated. In a filter, this predicate indicates under what circumstances a value from the temporal input is included in the temporal output.

Both of these situations are handled by forming an **XCASE** construct with two predicate-object pairs. One of the pairs is formed by associating the predicate with the object assertion that is associated with the corresponding temporal input. The other predicate-object pair is formed by associating the negation of the predicate with the special primitive object **UNDEFINED**. Note that **XCASE**s formed in this way have the same properties as **XCASE**s formed in conditionals. Clearly, given any predicate and its negation, exactly one of them will be true. Also note that in the case of a temporal output of a termination, this assertion form assumes that the termination predicate is such that once it is TRUE for some value in the input stream, it is TRUE for all remaining values in the input stream. This assumption is met by EOFP predicates (which are assumed to terminate all loops).

For example, suppose there is a filter with a temporal input that is associated with the assertion *obj1*, and which has a temporal output that corresponds to that temporal input and which is associated with the predicate *pred1*. Then an assertion will be made about the temporal output that is of the form:

```
(XCASE (pred1 obj1) ((NOT pred1) UNDEFINED))
```

In this way assertions are made about the temporal outputs of terminations and filters that state that, when a given predicate is true, the temporal data flow has a value that is the same as it had before it was operated on by the termination or filter. The assertions also state that when that predicate is not true, the temporal data flow has no value. Unfortunately, like the assertions produced for temporal output ports of augmentations, these assertions are indistinguishable from assertions representing single values.

5.4.4 Assertion Simplification Methods

It can be seen from the above discussion that there are only three types of assertions in the system; primitive objects, expressions composed of primitive functions (both arithmetic and boolean) acting on other objects, and **XCASE**s. The **XCASE**s are the only complex objects. Unfortunately, the way the assertions are built, **XCASE**s can appear in expressions and in predicate-object pairs of other **XCASE**s. This causes unnecessary complexity in all the assertion types.

All assertions in the system are kept as simple as possible by transforming the ones that contain **XCASE**s as components so that either the **XCASE** is eliminated or the **XCASE** is at the top level only. This is done for each assertion that is made in the system. This means that when a new assertion is formed, **XCASE**s can be nested at most one level deep in the assertion. This fact is used in the simplification process.

The transforms are such that predicates are often built which are not in simplest terms. These predicates are simplified through the use of a disjunctive normal form predicate simplifier that was designed by Deepak Kapur [12]. This predicate simplifier lies at the heart of the assertion simplification process.

5.4.4.1 Simplification of Predicate Assertions

When a predicate assertion is first formed in the system, it may contain an **XCASE** as an argument to a boolean function. Since an **XCASE** is an object, it will never appear as an argument to AND, OR, or NOT, but can only appear as an argument to comparative function such as EQUAL, GREATERP, etc. What is needed is a transform that will eliminate the **XCASE** from the expression. The transform that is used is given in Figure 34.

In this example, the second argument to a comparative function is an **XCASE**. It is assumed that this **XCASE** is already simplified. This means that *obj1* through *objn* are not **XCASEs**. Note that if the first argument to the comparative function had also been an **XCASE**, then the same transform could have been applied to each of the clauses that were produced in the first application of the transform, thereby eliminating all **XCASEs** from the expression. An inspection of Figure 34 should reveal that the resultant predicate has the intended truth value.

Once the transform has been applied, the expression is further simplified. The disjunctive normal form predicate simplifier does not know about the type of primitive objects that a given predicate will be expressed in terms of. Therefore, before the predicate is passed to the simplifier, it undergoes a prepass in which some of the subexpressions that are composed of a comparative function operating on two constants are replaced by TRUE or FALSE. For example,

```
(EQUAL (STRING abc)(STRING def))
```

can be replaced with FALSE.

Fig. 34. Transform to Remove XCASEs in Comparative Functions

```
              (comparative-function obj (XCASE (pred1 obj1)
                                                (pred2 obj2)
                                                (predn objn)))

Becomes:

              (OR (AND pred1 (comparative-function obj obj1))
                  (AND pred2 (comparative-function obj obj2))
                  (AND predn (comparative-function obj objn)))
```

In addition, any subexpression that involved a comparative function in which one of the arguments is the constant **UNDEFINED** is replaced with FALSE. This replacement is done because objects can be undefined but predicates cannot. Note that this replacement with FALSE (followed by simplification) is equivalent to first converting the **XCASE** to one in which *at most* one of the predicates is TRUE by removing the predicate-object pair in which the object is **UNDEFINED** (if any), and then performing the transform to eliminate the **XCASE** from the comparative function.

The expression is then passed to the general predicate simplifier. The result is a disjunctive normal form in which the clauses are as simple as possible and are in a canonical order.

As an example of the use of the predicate simplification transforms, let use consider the predicate in the termination of the temporal composition in PAYROLL2 (see Section 1.4). In this example, a flag is used to store the information about whether or not end of file has been reached. Because COBOL allows 88 variables to be used, flags of this type are very common in COBOL programs. Let us simplify the example by considering only the portion of the termination test that tests whether end of file has been reached on the HOURLY-WAGE file. The actual predicate in the analyzed plan checks for the value of the flag. The initial expression for the predicate as well as the final assertion actually formed for the predicate are shown in Figure 35.

5.4.4.2 Simplification of Object Assertions

The first transform for object assertions is used to simplify arithmetic expressions. Arithmetic expressions, as initially formed, can contain **XCASEs** as arguments to arithmetic functions such as TIMES and PLUS. What is needed is a transform that will change arithmetic expressions that contain **XCASEs** into an **XCASE** in which the objects of the predicate-object pairs are arithmetic expressions

Fig. 35. Example Predicate Simplification from PAYROLL2

```
The expression before simplification is:
(EQUAL 1. (XCASE ((EOFP (SEFO HOURLY-WAGE-IN)) 1.)
                 ((NOT (EOFP (SEFO HOURLY-WAGE-IN))) 0.)))

The expression after transform to eliminate the XCASE:
(OR (AND (EOFP (SEFO HOURLY-WAGE-IN)) (EQUAL 1. 1.))
    (AND (NOT (EOFP (SEFO HOURLY-WAGE-IN))) (EQUAL 1. 0.)))

The expression after the prepass:
(OR (AND (EOFP (SEFO HOURLY-WAGE-IN)) TRUE)
    (AND (NOT (EOFP (SEFO HOURLY-WAGE-IN))) FALSE))

The final assertion after simplification:
(EOFP (SEFO HOURLY-WAGE-IN))
```

that do not contain **XCASE**s. This keeps the **XCASE** forms at top level instead of nested within the arithmetic expressions. The transform that is used is shown in Figure 36.

In this example, the second argument to a binary arithmetic function is an **XCASE**. As before, it is assumed that the **XCASE** was already simplified and that *obj1* through *objn* are, therefore, not **XCASE**s. Had the first argument also been an **XCASE**, then the same transform could be applied to each of the objects in the predicate-object pairs resulting from the first application of the transform. The result is an expression that will have **XCASE**s nested at most one level deep, but in which none of the arithmetic expressions contain an **XCASE**s. The nested **XCASE**s, if any, are later removed with another transform.

The arithmetic expressions that result from the transform shown in Figure 36 can be reduced further by replacing any subexpression that contains an arithmetic function in which the constant **UNDEFINED** is used as an argument with the constant **UNDEFINED**. For example,

(TIMES *arg1* **UNDEFINED**)

is replaced with **UNDEFINED**.

XCASEs that are nested one level deep can result in two ways. One is by the application of the transform discussed above. The other occurs when a conditional segment is nested within an action of another conditional segment in the analyzed plan. In either case, it is desirable to eliminate the nested **XCASE**. If this were not done, then **XCASE**s nested to an arbitrary depth would eventually be formed. The transform that is used to eliminate nested **XCASE**s is shown in Figure 37.

In this example, one of the objects in a predicate-object pair of an **XCASE** is another **XCASE**. It is assumed that this nested **XCASE** is already simplified and that, therefore, *obj21* through *obj2n* do not contain **XCASE**s. Note that the same transform can be applied to any of the predicate-object pairs in the top level **XCASE** in which the object is an **XCASE**. The result of applying this transform is an **XCASE** that contains no nested **XCASE**s anywhere in the objects of the predicate-object pairs. An

Fig. 36. Transform to Remove XCASEs in Arithmetic Expressions

```
              (arithmetic-function obj (XCASE (pred1 obj1)
                                              (pred2 obj2)
                                              (predn objn)))
```

Becomes:

```
              (XCASE (pred1 (arithmetic-function obj obj1))
                     (pred2 (arithmetic-function obj obj2))
                     (predn (arithmetic-function obj objn)))
```

Fig. 37. Transform to Eliminate Nested XCASEs

```
(XCASE (pred11 obj11)
       (pred12 (XCASE (pred21 obj21)
                      (pred22 obj22)
                      (pred2n obj2n)))
       (pred1n obj1n))
```

Becomes:

```
(XCASE (pred11 obj11)
       ((AND pred12 pred21) obj21)
       ((AND pred12 pred22) obj22)
       ((AND pred12 pred2n) obj2n)
       (pred1n obj1n))
```

examination of Figure 37 should reveal that the resultant **XCASE** has the same semantics as the initial **XCASE**.

There is one remaining transform that can be applied to **XCASE**s. This transform is used when two or more predicate-object pairs contain the same object. Such an **XCASE** contains more predicate-object pairs than is necessary. In this event, the number of predicate-object pairs can be reduced by applying the transform shown in Figure 38. This transform can be used to condense all sets of predicate-object pairs that contain the same object. The result is an **XCASE** in which all of the objects are distinct. In order to see that this transform retains the semantics of the initial **XCASE**, it must be recalled that the predicates in the initial **XCASE** have the property that exactly one of them will be true at a time. Therefore, the ORs that are formed have as arguments a set of predicates in which at most one of them is true. This property ensures that the resultant **XCASE** is equivalent to the initial one.

Fig. 38. Transform to Condense Predicate-object Pairs containing Identical Objects

```
(XCASE (pred1 obj1)
       (pred2 obj1)
       (predn objn))
```

Becomes:

```
(XCASE ((OR pred1 pred2) obj1)
       (predn objn))
```

After all of the above transforms are applied to a given **XCASE**, then each of the predicates in the predicate-object pairs is passed to the disjunctive normal form predicate simplifier. Some of the predicates may be identically FALSE. In this case, the predicate-object pair containing that predicate is simply removed from the **XCASE**. Removal of predicate-object pairs in this fashion can result in an **XCASE** in which all predicate-object pairs except one have been removed, and the single remaining pair may contain a predicate that is identically TRUE. In this event, the entire **XCASE** is eliminated and the object of the last remaining pair is used as the final form of the assertion.

In all cases, the final result of the application of all the transforms discussed above is an assertion for an object that is either a primitive object, an arithmetic expression in simplest form that contains no **XCASE**s, or an **XCASE** in which all of the predicates are in canonical form and do not contain **XCASE**s, and all of the objects are in simplest form and do not contain **XCASE**s.

For example, consider the simplification steps taken to simplify an expression for GROSS-PAY, taken from PAYROLL2 (see Section 1.4) shown in Figure 39. This expression is first built at the end of the conditional that checks to see if the key fields are equal before calculating GROSS-PAY.

Because this simplification is done to each assertion before it is added to the plan for the program, all object assertions in the system are always in simplest form. This is not only a great aid in debugging, but also ensures that the expressions that are passed on to the HIBOL production phase are as simple as possible.

5.4.5 Communication between Symbolic Evaluation and HIBOL Production

As indicated above, when a CWRITE or CREWRITE function is evaluated in the symbolic evaluation of the analyzed plan, the assertions that had been formed for the non-file-object arguments to the write function are stored so they can be used in the HIBOL production phase. However, just this information is not quite enough. It is also necessary to store the control environment in which the write function is evaluated.

The control environment is kept in a stack that is manipulated during the symbolic evaluation. Each time an action of a conditional is evaluated, the predicate that determines under what conditions that action will take place is pushed on the stack. The stack is then popped after the evaluation of that action is complete.

In addition, within a temporal composition, it is sometimes the case that certain augmentations receive dummy temporal data flow from a termination or filter. These dummy temporal data flows do not contain any data values, but simply cause the augmentation to only be executed when the predicate of the termination or filter with which they are associated is true. To take this fact

Fig. 39. Example Simplification of an Object Assertion

Expression initially formed:

```
(XCASE ((EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
               (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))
        (TIMES (XCASE ((NOT (EOFP (SEFO HOURLY-WAGE-IN)))
                       (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE))
                      ((EOFP (SEFO HOURLY-WAGE-IN))
                       UNDEFINED))
               (XCASE ((NOT (EOFP (SEFO HOURS-WORKED-IN)))
                       (CREAD-VAL HOURS-WORKED-IN HOURS-WORKED))
                      ((EOFP (SEFO HOURS-WORKED-IN))
                       UNDEFINED))))
       ((NOT (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                    (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER)))
        UNDEFINED))
```

Expression after simplification of arithmetic expression:

```
(XCASE ((EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
               (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))
        (XCASE ((AND (NOT (EOFP (SEFO HOURLY-WAGE-IN)))
                     (NOT (EOFP (SEFO HOURS-WORKED-IN))))
                (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
                       (CREAD-VAL HOURS-WORKED-IN HOURS-WORKED)))
               ((OR (EOFP (SEFO HOURLY-WAGE-IN))
                    (EOFP (SEFO HOURS-WORKED-IN)))
                UNDEFINED)))
       ((NOT (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                    (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER)))
        UNDEFINED))
```

Expression after transform to eliminate nested XCASE:

```
(XCASE ((AND (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                    (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))
             (AND (NOT (EOFP (SEFO HOURLY-WAGE-IN)))
                  (NOT (EOFP (SEFO HOURS-WORKED-IN)))))
        (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
               (CREAD-VAL HOURS-WORKED-IN HOURS-WORKED)))
       ((AND (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                    (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))
             (OR (EOFP (SEFO HOURLY-WAGE-IN))
                 (EOFP (SEFO HOURS-WORKED-IN))))
        UNDEFINED)
       ((NOT (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                    (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER)))
        UNDEFINED))
```

Fig. 39. Example Simplification of an Object Assertion (Continued)

Expression after condensing clauses with identical objects:

```
(XCASE ((AND (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                    (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))
             (AND (NOT (EOFP (SEFO HOURLY-WAGE-IN)))
                  (NOT (EOFP (SEFO HOURS-WORKED-IN)))))
        (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
               (CREAD-VAL HOURS-WORKED-IN HOURS-WORKED)))
       ((OR (AND (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                        (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))
                 (OR (EOFP (SEFO HOURLY-WAGE-IN))
                     (EOFP (SEFO HOURS-WORKED-IN))))
            (NOT (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                         (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))))
        UNDEFINED))
```

Final assertion after predicate simplification:

```
(XCASE ((AND (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                    (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))
             (NOT (EOFP (SEFO HOURLY-WAGE-IN)))
             (NOT (EOFP (SEFO HOURS-WORKED-IN))))
        (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
               (CREAD-VAL HOURS-WORKED-IN HOURS-WORKED)))
       ((OR (EOFP (SEFO HOURLY-WAGE-IN))
            (EOFP (SEFO HOURS-WORKED-IN))
            (NOT (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                        (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER)))))
        UNDEFINED)
```

into account, when an augmentation that has a dummy temporal input is symbolically evaluated, the predicate that is associated with that dummy input is pushed onto the control environment stack. The stack is then popped when evaluation of that augmentation is complete.

Within temporal compositions, there is an additional implicit factor in the control environment. Recall that an augmentation within a temporal composition is only executed if none of the terminations in the temporal composition have caused the loop to terminate. Therefore, the negation of the predicates that cause the loop to terminate must be considered part of the control environment.

The control environment of write functions is saved through the following mechanisms. A list is kept of all temporal compositions in the plan along with the predicates that cause each to terminate. When a write function is symbolically evaluated, the control environment stack is saved as well as the

Fig. 40. Information Transferred Between Phases in PAYROLL

```
Temporal Composition: TEMPCOMP-1
Termination Predicate: (EOFP (SEFO HOURLY-WAGE-IN))
Output Expressions:
(GROSS-PAY-REC_EMPLOYEE-NUMBER IS
      (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE-REC_EMPLOYEE-NUMBER))
(GROSS-PAY IS (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE) 40.))

Temporal Composition: OUTPUT-NOT-IN-TEMPCOMP
Termination Predicate: NIL
Output Expressions:
(EMPLOYEE-COUNT IS (COUNT (NOT (EOFP (SEFO HOURLY-WAGE-IN)))))
(TOTAL-GROSS-PAY IS
      (SUM (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE) 40.)))

Filename: HOURLY-WAGE-IN
Open Type: COPENI

Filename: GROSS-PAY-OUT
Opentype: COPENO

Filename: EMPLOYEE-COUNT-OUT
Opentype: COPENO

Filename: TOTAL-GROSS-PAY-OUT
Opentype: COPENO
```

name of the temporal composition in which it is located. Write functions not located within temporal compositions are associated with a special dummy temporal composition called OUTPUT-NOT-IN-TEMPCOMP.

Before the termination of the symbolic evaluation phase, the list of temporal composition names and their associated predicates as well as the information stored during the evaluation of write functions are stored in a file to be used in the HIBOL production phase. The only remaining information that is transferred to the HIBOL production phase is the type of open function that was used to open each file in the COBOL program. As an example, the information transferred from the symbolic evaluation phase to the HIBOL production phase in the translation of PAYROLL is shown in Figure 40.

5.5 DATA DIVISION Query

A file is input to the translation process directly from the COBOL parser that contains virtually all of the information that appears in the DATA DIVISION of the COBOL program. Included is the structure of the buffer area associated with each file as well as the PICTURE clause for each atomic variable name in these structures. The only needed information that is not included in this file, because it does not appear anywhere in the COBOL program, is which of the atomic variable names in the buffer structure for each file are key fields, and in which order those key fields were used to sort the file. An exception occurs when a file is specified in the DATA DIVISION of the COBOL program to be used for random access. In this case, the syntax and semantics of COBOL demand that the needed information about key fields and sort order be explicitly given in the DATA DIVISION. However, the current implementation of the translation process does not handle random access files.

This information is gathered by simply asking the user of the SATCH system to supply it. For each file, a list of the atomic variable names of the corresponding buffer structure is displayed on the screen along with associated numbers. The user then simply types in the list of numbers that correspond to the key fields in the order that they were used to sort the file. This information is then added to the file of DATA DIVISION information to be used in the HIBOL production phase.

The fact that this information needs to be gathered from the SATCH system user is not a major drawback of the system. Anyone that is at all familiar with the files that are used in a production COBOL system should at least know which fields in each file are key fields even if they do not know what the particular program in question is doing.

5.6 HIBOL Production

The information gathered in the analyzed plan symbolic evaluation and the data division query is used to produce the actual HIBOL for the COBOL program. This process is further subdivided into two subprocesses; one which produces the DATA DIVISION of the HIBOL program, and one which produces the COMPUTATION DIVISION.

The subprocess that produces the DATA DIVISION of the HIBOL program is relatively trivial. The names of the key fields, gathered in the data division query, as well as the information about the corresponding PICTURE clauses, received directly from the COBOL parser, are used to produce the KEY SECTION. The information about the type of OPEN function used for each file, gathered in the analyzed plan symbolic evaluation, and the information about the buffer-structure and corresponding data and key fields, received directly from the COBOL parser, are used to produce the INPUT and OUTPUT SECTIONs. Each data field name in the buffer-structure for every file in the COBOL program is made into an individual data-set in either the INPUT SECTION or OUTPUT SECTION

Fig. 41. Steps in the Production of the COMPUTATION DIVISION

1. Remove assertions for key fields from further consideration.
2. Add to each assertion the negation of the predicates that terminate the temporal composition in which they were formed.
3. Remove predicate-object pairs with an object that is **UNDEFINED** from **XCASE**s.
4. Consolidate the assertions for a given output data field formed in separate temporal compositions into one assertion.
5. Replace EOFP and comparative predicates with **FILE-PRESENT** predicates.
6. Eliminate **FILE-PRESENT** predicates that are redundant with the semantics of HIBOL.
7. Convert object assertions into HIBOL syntax.
8. Replace any remaining **FILE-PRESENT** predicates with PRESENT predicates.
9. Convert predicate assertions into HIBOL syntax.
10. Output final HIBOL expressions into HIBOL file.

depending on whether the OPEN function used on the file was COPENI or COPENO. A data-set is created in both sections if the file was opened via COPENIO. Currently, a VARIABLE SECTION is never used.

The subprocess that produces the COMPUTATION DIVISION of the HIBOL program is much more complex. The largest difficulty in performing this task is the determination of the correct predicates to be used in the conditional expressions. Therefore, this subprocess consists mainly of the manipulation of various predicates in various ways, starting from the assertions received from the symbolic evaluation of the analyzed plan. An overview of the steps performed in the production of the COMPUTATION DIVISION is given in Figure 41.

The first four steps result in a single conditional assertion for every data field of every output file. These assertions will be in one-to-one correspondence with the desired output data-set definitions that will appear in the final HIBOL program. The next six steps convert each of the resultant assertions into the corresponding output data-set definition.

5.6.1 Remove Key Field Assertions from Further Consideration

First, all assertions for key fields are dropped at this point and not processed further. The key field expressions are dropped because the HIBOL COMPUTATION DIVISION does not contain expressions for key fields. It is safe to drop them because, based on our assumptions about the type of COBOL programs being processed, the assertions for the key fields are controlled by the same basic predicates that control the data fields and, therefore, no needed information is contained within them.

5.6.2 Assert Negation of Termination Predicates

Then, for each remaining assertion that was produced in a temporal composition, it is asserted that the assertion holds whenever the predicates that would cause the temporal composition to terminate are FALSE. This is done by forming an **XCASE** with two predicate-object pairs. The first pair consists of the negation of the logical OR of the predicates that cause the temporal composition to terminate and the original assertion for the object, and the second pair consists of the logical OR of those same predicates combined with the object **UNDEFINED**.

Consider an example from PAYROLL. The original assertion associated with the variable GROSS-PAY is:

```
(TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE) 40.)
```

and the predicate causing the temporal composition to terminate is:

```
(EOFP (SEFO HOURLY-WAGE-IN))
```

The **XCASE** that would be produced is:

```
(XCASE ((NOT (EOFP (SEFO HOURLY-WAGE-IN)))
         (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE) 40.))
        ((EOFP (SEFO HOURLY-WAGE-IN)) UNDEFINED))
```

This **XCASE** would then be simplified using all of the simplification techniques discussed in previous sections. Note that if the original assertion had already been an **XCASE**, then this process would have the effect of ANDing the predicate in every predicate-object pair with the negation of the predicates that terminate the temporal composition. A further effect of this transformation is that all assertions formed within temporal compositions are now **XCASEs**.

5.6.3 Remove UNDEFINED from XCASEs

Next, the predicate-object pair of the XCASE, if any, that has an object of UNDEFINED is removed from the XCASE. The resultant XCASE no longer has the property that *exactly one* of the predicates will be true at a time, but still has the property that *at most one* of the predicates will be true at a time. It is safe to remove these pairs for two reasons. First, from this point on in the translation, no transform will be applied to the XCASEs which requires that the predicates be all inclusive, although transforms will be applied that require that they be mutually exclusive. Second, the semantics of HIBOL conditional statements (into which the XCASEs will be translated) state that, for any given element in the key space, if none of the predicates in the conditional are TRUE, the conditional will be undefined for that element, and the element will not be included in the actual key space of the result. In addition, these predicate-object pairs need to be eliminated at this time so that the next operation to be performed on the assertions will function properly.

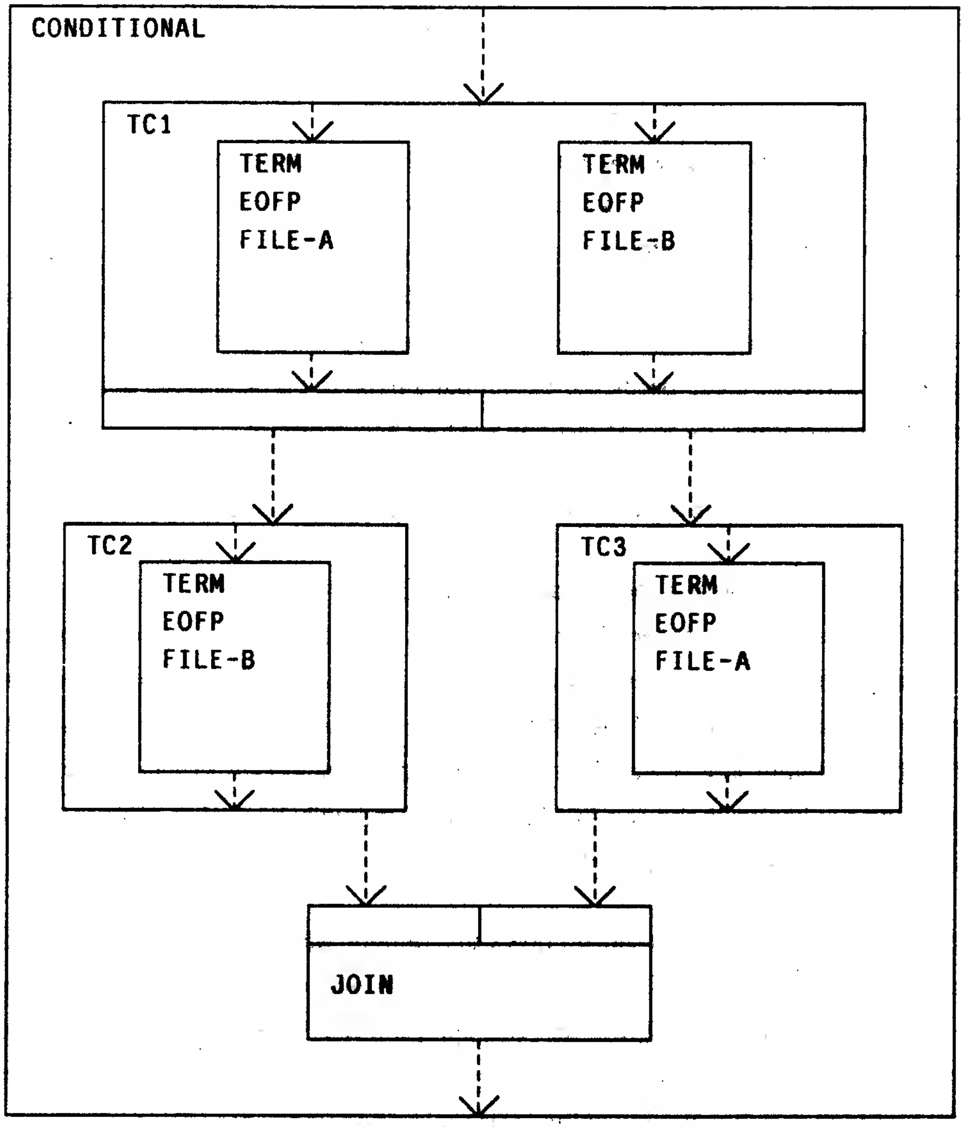
5.6.4 Consolidate Different Assertions for the Same Data Field Into One

The last thing that is done to produce a single conditional expression for every data field of every file is to look for assertions for a particular data field in more than one temporal composition. If more than one assertion is found for a given data field, the predicate-object pairs of one are simply appended to the predicate-object pairs of the other, forming a larger XCASE which is then simplified. It is important that the resultant XCASE have the same predicate exclusivity of all other XCASEs. For this to be the case, the predicates in the two XCASEs must be mutually exclusive. This will, in turn, be true if the initial COBOL program adheres to the current assumptions of the system.

This transform is necessary to translate programs (among others) which perform a file merge operation. (See the LOC-LIST example in Section 1.4). A high level view of the typical analyzed plan for a file merge operation is shown in Figure 42. The analyzed plan is a conditional with a temporal composition acting as the predicate and two additional temporal compositions acting as the actions. Note that only the termination subsegments of the temporal compositions are shown.

A summary of the predicates that will be included in *every* predicate-object pair in an XCASE in each of the three temporal compositions is shown in Figure 43. (Recall that the predicates in these XCASEs are no longer all inclusive since the predicate-object pairs containing UNDEFINED have already been removed). These predicates are included in the XCASEs either because they are the negation of the loop terminator, in which case they were inserted by a previous step in the HIBOL production phase as described above, or because they were on the control stack when the temporal composition was symbolically evaluated, in which case they already appeared in the assertions transferred from that phase to the HIBOL production phase. By examining this figure, it should be easy to see that these predicates are in fact mutually exclusive and that, therefore, the transform

Fig. 42. Sketch of Analyzed Plan for File Merge Operation



functions properly in this example.

5.6.5 Replace EOFP and Comparative Predicates with FILE-PRESENT Predicates

The next transform replaces all EOFP predicates and all comparative functions applied to key fields by FILE-PRESENT predicates. The replacement scheme is shown in Figure 44. The replacement for EOFP predicates should be fairly obvious.

Fig. 43. Predicates Contained in XCASEs in a File Merge

```
TEMPCOMP-1:  (AND (NOT (EOFP (SEFO FILE-A)))
                  (NOT (EOFP (SEFO FILE-B))))

TEMPCOMP-2:  (AND (EOFP (SEFO FILE-A))
                  (NOT (EOFP (SEFO FILE-B))))

TEMPCOMP-3:  (AND (NOT (EOFP (SEFO FILE-A)))
                  (EOFP (SEFO FILE-B)))
```

Fig. 44. Replacement Predicates

```
(EOFP (SEFO file-name))
Becomes:
(NOT (FILE-PRESENT file-name))

(EQUAL (CREAD-VAL file-name1 key-field-name)
       (CREAD-VAL file-name2 key-field-name))
Becomes:
(OR (AND (FILE-PRESENT file-name1)
         (FILE-PRESENT file-name2))
    (AND (NOT (FILE-PRESENT file-name1))
         (NOT (FILE-PRESENT file-name2))))

(LESSP (CREAD-VAL file-name1 key-field-name)
       (CREAD-VAL file-name2 key-field-name))
Becomes:
(AND (FILE-PRESENT file-name1)
     (NOT (FILE-PRESENT file-name2)))

(GREATERP (CREAD-VAL file-name1 key-field-name)
          (CREAD-VAL file-name2 key-field-name))
Becomes:
(AND (NOT (FILE-PRESENT file-name1))
     (FILE-PRESENT file-name2))
```

The replacements for the comparative functions, however, are less obvious. It should first be noted that in order for the replacement to be performed, it must be the case that the key fields that are acted on by the comparative function must be the same. Currently, two key fields from different files are considered the same if they have the same name. Later, a suggestion is made as to how this constraint could be relaxed.

The replacement predicates for comparative functions on key fields rely heavily on the assumption that the program in which they are formed is one of the three basic types, and that the two files under consideration are being read sequentially and are sorted in key field order. The number of key fields, however, is unimportant. Let us look more closely at these replacement predicates. If the value of the key field read from `file-name-1` is less than the value of the key field read from `file-name-2`, then that means that the record in `file-name-2` corresponding to the key value read in `file-name-1` is missing while it does appear in `file-name-1`. On the other hand, if the value of the key field read from `file-name-1` is greater than the value of the key field from `file-name-2`, then that means that the record in `file-name-1` that corresponds to the key value read in `file-name-2` is missing while it does appear in `file-name-2`. These facts are reflected in the replacement predicates for LESSP and GREATERP respectively.

If the values of the key fields read from both files are equal, then both records appear for that key value. This fact is reflected in the first clause of the replacement predicate for EQUAL. The second clause of the replacement predicate for EQUAL is included so that the the replacement predicates considered as a whole will exhibit a very useful property. Namely, they exhibit all of the tautologies that are exhibited by GREATERP, LESSP, EQUAL and NOT-EQUAL. For example, after simplification with the disjunctive normal form predicate simplifier, the predicate produced from

```
(NOT (LESSP (CREAD-VAL file-name1 key-field-name)
            (CREAD-VAL file-name2 key-field-name)))
```

should be logically equivalent to the predicate produced from

```
(OR (GREATERP (CREAD-VAL file-name1 key-field-name)
              (CREAD-VAL file-name2 key-field-name))
    (EQUAL (CREAD-VAL file-name1 key-field-name)
           (CREAD-VAL file-name2 key-field-name)))
```

both before and after the replacement has been made. The replacement predicates currently used do produces the equivalent result. Both the predicates shown above, after replacement, reduce to

```
(OR (NOT (FILE-PRESENT file-name-1))
    (FILE-PRESENT file-name-2))
```

The fact that the replacement predicates exhibit this property eliminates the possibility that different predicates could be produced after replacement simply because the programmer of the original COBOL program chose a particular form for a predicate over an equivalent form.

As an example of the use of predicate replacement, consider the expressions, taken from PAYROLL2 (see Section 1.4), for GROSS-PAY both before and after predicate replacement as shown in Figure 45. Note that after the replacement, the resultant predicates are simplified.

Fig. 45. Example of Predicate Replacement

```
Expression for GROSS-PAY before replacement:
(XCASE ((AND (EQUAL (CREAD-VAL HOURLY-WAGE-IN EMPLOYEE-NUMBER)
                    (CREAD-VAL HOURS-WORKED-IN EMPLOYEE-NUMBER))
             (NOT (EOFP (SEFO HOURLY-WAGE-IN)))
             (NOT (EOFP (SEFO HOURS-WORKED-IN))))
        (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
               (CREAD-VAL HOURS-WORKED-IN HOURS-WORKED))))

Expression for GROSS-PAY after replacement:
(XCASE ((AND (FILE-PRESENT HOURLY-WAGE-IN)
             (FILE-PRESENT HOURS-WORKED-IN))
        (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
               (CREAD-VAL HOURS-WORKED-IN HOURS-WORKED))))
```

5.6.6 Eliminate Redundant FILE-PRESENT Predicates

The next transform eliminates the **FILE-PRESENT** predicates that are redundant with the semantics of HIBOL. The **FILE-PRESENT** predicates in the predicate of each predicate-object pair that refer to the same file as any of the remaining **CREAD-VAL** objects in either the predicate or object of that particular predicate-object pair are eliminated by replacing them with TRUE, and then simplifying the predicate. These predicates are redundant with the semantics of HIBOL because all HIBOL expressions contain an implicit PRESENT predicate for every data-set name that appears in the expression. All predicate-object pairs with a resultant predicate of FALSE are dropped from the **XCASE**. It often happens that the resultant **XCASE** has only a single predicate-object pair with a predicate of TRUE. If this occurs, the **XCASE** is reduced to the object of that predicate-object pair.

Continuing the example from PAYROLL2 shown in Figure 45, the expression for GROSS-PAY at this point in the processing is simply:

```
(TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
       (CREAD-VAL HOURS-WORKED-IN HOURS-WORKED))
```

5.6.7 Express Objects In HIBOL Syntax

Next, the object portion of each predicate-object pair as well as those object expressions that are not part of **XCASEs** are transformed into HIBOL syntax. Several things must be done. First, all of the arithmetic operation expressions are converted from prefix to fully parenthesized infix form. Second, certain forms are converted to match the HIBOL syntax. For example, TIMES is converted to "*" and STRING objects are converted into character strings. Third, **CREAD-VAL** expressions are converted into the appropriate data-set references. If the file referred to in the **CREAD-VAL** expression is opened for input then the resultant expression is simply the data-set name that corresponds to the data field. If the file is opened for input-output, then the resultant expression is:

```
(LAST PERIOD'S data-set-name)
```

to reflect the fact that the data-set name refers to the data-set in the INPUT SECTION and not the data-set in the OUTPUT SECTION. (See the DBINIT example in Section 1.4.)

5.6.8 Replace Remaining FILE-PRESENT Predicates with PRESENT Predicates

Next, any remaining **FILE-PRESENT** predicates are replaced with PRESENT predicates acting on data-set names. If the file that the **FILE-PRESENT** predicate refers to has a single data field, then the data-set name that corresponds to that data field is used as the argument to the PRESENT predicate. However, if the file has more than one data field, then there is no way to automatically determine which data-set name(s), corresponding to particular data field(s), should be used in PRESENT predicate(s). From the perspective of the COBOL program, it does not matter because if any of the data fields are present for a given index, then all of the data fields will be present for that index. But, from the perspective of the HIBOL, all of the data fields for a given COBOL file have each been given an individual data-set name and the information that dictates that if one is present they all are present has been lost.

On the other hand, it is usually not desirable to demand that all of the data-sets that correspond to the original COBOL data fields for the file be included in PRESENT predicates in the HIBOL. Therefore, the user of the SATCH system is queried to determine which of the possible PRESENT predicates acting on data-set names should be included. This process is simplified by the fact that the objects of the predicate-object pairs have already been converted to HIBOL syntax, and therefore can be shown to the user in a more readable form. The user is shown the HIBOL for the object in the predicate-object pair as well as a list of the data fields for each of the files included in a **FILE-PRESENT** predicate, and asked to supply a list of data field names for which PRESENT predicates should be formed. These PRESENT predicates are then formed and placed into the predicates of the predicate-object pairs in place of the **FILE-PRESENT** predicates which are then simplified.

5.6.9 Express Predicates in HIBOL Syntax

The resultant predicates are now converted into HIBOL syntax. This is very similar to the conversion to HIBOL of the object expressions. One difference is that the logical functions AND and OR are n-ary operators. Therefore, when they are converted into infix notation, copies of the operator are placed between every two operands. In addition, PRESENT predicates acting on data fields from a file opened via COPENIO are converted into

```
(LAST PERIOD'S field-name PRESENT)
```

instead of the usual PRESENT predicate. (See the DBINIT example in Section 1.4).

5.6.10 Output Final HIBOL for COMPUTATION DIVISION

The last step in the production of the COMPUTATION DIVISION of the HIBOL program consists of outputting the expressions into the HIBOL file. This consists of outputting an expression for every data-set defined in the OUTPUT SECTION. The name of the output data-set is followed by "IS" and then followed by the HIBOL expression produced above. **XCASE**s are processed by running through the predicate-object pairs first outputting the expression for the object and then the one for the predicate, inserting IF and ELSE in the appropriate places. Currently, little effort has been spent to get the indentation of conditional expressions correct, and the examples shown in Section 1.4 have been reformatted by hand.

6. Critique of the Current Implementation of the Translation Process

In the first part of this chapter, several problems that arise in the current implementation of the translation process will be discussed, and suggestions made as to how they might be eliminated. In the second part of this chapter, suggestions are made as to how the translation process could be expanded to handle a larger domain of COBOL programs.

6.1 Problems Arising in the Current Implementation

Perhaps the most glaring problem with the current implementation of the translation process is that it does not recognize when it has gone astray. It blindly assumes that the program with which it is dealing adheres to all the implicit restrictions. If the program does not adhere to the appropriate assumptions, the program will still try to produce some HIBOL program even though it probably will not be correct. Obviously, a more robust system needs to recognize when it is given a COBOL program that it cannot translate and then act accordingly. Later in this chapter, a few minor suggestions are made as to how this problem could be somewhat reduced although not eliminated. In the next chapter, a suggestion is made about a second generation system that would significantly reduce this problem, if not eliminate it.

The remainder of this section discusses four more specific problem areas in the current implementation of the translation process. For some of the problem areas, satisfactory solutions are proposed. For others, no satisfactory solutions have yet been determined, although partial solutions are proposed. The first subsection discusses issues concerning the assertions formed in the symbolic evaluation of the analyzed plan. The second subsection discusses the issue of variable names, and how more mnemonic HIBOL code can be produced by the renaming of variables. The third subsection discusses the problems encountered in the production of readable HIBOL code for count operations. The last subsection discusses the issue of the use of output data-set names on the right hand side of data-set definitions in the COMPUTATION DIVISION of the HIBOL code.

6.1.1 Assertions Formed During the Symbolic Evaluation

One problem with the current method used to form assertions during the symbolic evaluation of the analyzed plan is that the assertions formed for temporal data ports are indistinguishable from those formed for non-temporal ports. The information that the temporal port contains a stream of values instead of a single value is discarded. Therefore, the assertions formed for temporal data ports are not semantically correct.

Assertions that are semantically correct could be formed for temporal outputs of augmentations by including in the assertion the information that the object is in fact a stream as well as the information that indicates for which values of a predicate values in the stream are defined. Temporal outputs from augmentations obtain their values from two different places relative to the augmentation function: from an output of the augmentation function or from an input to the augmentation function. (See the generating augmentation from PAYROLL shown in Figure 27.) These two cases have similar, but slightly different, semantics. Let us examine these two cases in more detail.

Temporal outputs that obtain their values from an output of the augmentation function represent streams of values in which all of the values are produced the same way, via the augmentation function. These streams have the additional characteristic that they are truncated at a point that is determined by the predicates that control the termination subsegments of the temporal composition in which they appear. Therefore, semantically correct assertions for these streams must contain three pieces of information. First, the assertions should indicate that they do in fact refer to a stream, and not a single value. Second, they should impart the notion that all of the values in the stream follow the same prototype. Third, they should include a predicate that indicates under what circumstances the values in the stream are defined. This predicate is the conjunction of the negations of the predicates that terminate the loop.

For example, in PAYROLL, the temporal output that contains values for HOURLY-WAGE is currently given the following assertion:

```
(CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE)
```

However, a more semantically correct assertion might be:

```
(FOR-ALL-TRUE-OCCURRENCES-OF (NOT (EOFP (SEFO HOURLY-WAGE-IN)))
                             (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE))
```

The inclusion of the old object assertion within the **FOR-ALL-TRUE-OCCURRENCES-OF** clause indicates that the object is in fact a stream, that all of the values in that stream follow the **CREAD-VAL** prototype, and that it has defined values until the end of file is reached on **HOURLY-WAGE-IN**.

Temporal outputs that obtain their values from an input to the augmentation function have the same semantics as temporal outputs that obtain their values from an output of the augmentation function except that the first value in the stream is different from the rest of the values in that it takes its value from either the initialization of the augmentation or from outside the augmentation altogether. Therefore, a semantically correct assertion for such a temporal output might be:

```
((FIRST-VALUE object1)
 (FOR-ALL-TRUE-OCCURRENCES-OF predicate object2))
```

Such an assertion indicates that the first value of the stream is distinct from the rest of the values of the stream, and therefore does not follow the same prototype. For example, a semantically correct assertion for the temporal output of the augmentation that performs the SUM operation in PAYROLL[1] would be:

```
((FIRST-VALUE 0.)
 (FOR-ALL-TRUE-OCCURRENCES-OF (NOT (EOFP (SEFO HOURLY-WAGE-IN)))
            (SUM (TIMES (CREAD-VAL HOURLY-WAGE-IN HOURLY-WAGE) 40.))))
```

Semantically correct assertions could be formed for the temporal outputs of terminations by using the forms described above. For example, the temporal output of the termination in PAYROLL (see Figure 28) would be

```
(FOR-ALL-TRUE-OCCURRENCES-OF (NOT (EOFP (SEFO HOURLY-WAGE-IN)))
                             DUMMY)
```

Assertions formed for the temporal outputs of filters need to incorporate the predicate assertion that corresponds to the out-case of the filter predicate with which they are associated as well as the predicates discussed above. This could be done by forming the logical AND of the filter predicate and the one which already appears in the input temporal flow in the FOR-ALL-TRUE-OCCURRENCES-OF clause. Using the filter example from a previous chapter (see Figure 29), the temporal output for the negative values could be given the assertion

```
(FOR-ALL-TRUE-OCCURRENCES-OF (AND (NOT stream-truncation-predicate)
                                  (LESSP num 0.))
                             num)
```

It should be stated that the assertion forms for temporal outputs described above are still based on the assumption that the termination predicates remain TRUE for all values in the input stream used as input to the termination after they are TRUE for some initial value. EOFP predicates have this property (and are assumed to terminate all loops).

Although the assertions for temporal data ports described above would be more semantically correct than the ones currently formed, they would be of limited use to the translation process. The main reason for this is that the augmentation functions that eventually consume the temporal flows, and in particular the augmentation functions that correspond to write functions with which we are especially interested, only have non-temporal inputs. The temporal flows arrive at the input temporal data ports of the augmentation, but are then decomposed into individual values before being passed

[1] Note that since the partial sums formed in this augmentation are not actually used anywhere, this temporal output does not appear in the diagrams for PAYROLL shown in a previous chapter.

on to the augmentation function. During this decomposition, the information that states that the object is a stream would have to be stripped back off, and an assertion formed that again represents a single value. This is because the augmentation function operates on a *typical* value of the input stream. Assertions that are currently formed exactly express the typical value that is desired.

In addition, use of the more complex assertions described above would call for the development of additional assertion simplification techniques to handle them. The number of simplification techniques required goes up as the cross-product of the number of different object types in the system. This fact creates a desire to limit the number of different object types as much as possible.

Another problem with the creation of the more complex assertions described above is that, because of the order in which subsegments of the temporal composition are symbolically evaluated, the predicates that terminate the loop are not generally known at the time that the assertions are formed for the temporal outputs of the generating augmentations. Further, it is not possible to change the order in which the subsegments are symbolically evaluated because the termination cannot be evaluated until something is known about the values of its inputs which are, in general, produced within some generating augmentation.

In summary, it would be difficult to produce more semantically correct assertions for temporal data ports, their inclusion would call for the development of more simplification techniques, and they would be of limited usefulness to the translation process. Therefore, the current implementation retains the loop termination information by storing the predicates that terminate each temporal composition, and associating with every assertion passed on to the HIBOL production phase the name of the temporal composition in which it was formed. This technique has proved adequate for the COBOL programs examined to date.

Another shortcoming of the current assertion technique is that assertions formed for file-objects do not contain any history of the operations that have been performed on them. This eliminates the ability to detect non-standard read sequences on the file that could skip over records or perform other forbidden operations. A more robust system would have to examine the sequence of operations performed on file-objects fairly closely in order to guarantee that the HIBOL produced is a correct translation of the initial COBOL program.

6.1.2 Variable Names

In the previous chapter, it was mentioned that currently key field names referring to the same key field in different files must be identical, and that structure names are sometimes added to data field names by the COBOL parser in order to make them unique. It is desirable to eliminate this constraint on key field names and to make the names used for both data and key fields more mnemonic.

The DATA DIVISION of a HIBOL program uses a single name for a particular key field no matter how many data-sets use that key field. This is not a feature that simply increases the readability of the HIBOL, but is demanded by the semantics of the language. Currently it is easy to produce HIBOL that conforms to this constraint as it is assumed that all key fields that refer to a particular key are given the same name in the COBOL program.

This constraint could be eliminated by the following change to the data division query subprocess. The key field query for the first file proceeds as always. Then, for each subsequent file, after the key fields and sort order have been given, a list of the currently known key fields is presented and the user is asked to make correspondences between the key fields in the current file and the key fields in the accumulated list. After all input files are processed in this manner, the sets of corresponding key fields are presented one at a time and the user asked to supply a mnemonic name that should be used for that key field in the final HIBOL code.

This process demands information from the user of the SATCH system that is no different in kind than that demanded by the current data division query. If the user is able to recognize which fields in a given file are key fields, then it should also be possible to recognize the same key field in different files.

The semantics of HIBOL demand, as one might expect, that all data-set names be unique. The exception occurs when an input and an output data-set have the same name and the HIBOL program performs an update operation on that data-set (see the DBINIT example in Section 1.4). The data field names given in a COBOL program, however, might not be unique, although the data field name together with the name of the structure that it is contained in is always unique. Currently, to avoid ambiguity, the COBOL parser always produces unique identifiers for data fields by adding the structure name when it is necessary to do so. In general, the data field names produced from the COBOL program might not be particularly mnemonic especially when the parser has to add the structure name.

It is possible to produce a HIBOL program that is much more readable and maintainable if the data-set names are given mnemonic names. The later is made easier by the fact that the HIBOL program is sufficiently abstract that each of the data-sets should correspond to some real world parameter in the system that the program is an implementation of. Therefore, it may be desirable to give the data-sets in the HIBOL program names that differ from the names for the data fields in the COBOL program to which they correspond.

These data-set names must be supplied by the SATCH system user. The user should not be expected to supply a data-set name without being shown a context in which that data-set will be used. However, it is undesirable to ever show the user of the SATCH system any expressions in the syntax of the assertions formed in the symbolic evaluation of the analyzed plan because, as is clear from the examples given above, it is cumbersome and difficult to read. Therefore, the best time to query the user of the SATCH system for data-set names is after the phase of the HIBOL production subprocess in which the expressions are converted into HIBOL syntax, but before the user is queried for the data-set names to include in PRESENT predicates used to replace the **FILE-PRESENT** predicates.

The user would be shown one expression at a time. As each expression was presented, the user would be asked to supply replacement names for each of the data fields that are referenced which have not already been given a data-set name. All data fields that have already been given a name by the user would appear as that new name. The process would be continued until all data fields had been given a data-set name.

It is not known exactly how difficult it will be for the user of the SATCH system to recognize the context that is presented for a given data field. Undoubtedly, this portion of the system will require some human engineering before the query process could proceed smoothly. It is hoped that, if properly engineered, this query process will not be too difficult for the user.

It should be noted that the above discussion, as well as the current system implementation, makes the implicit assumption that all the data fields in the various input and output files do in fact contain different information even though their names (minus structure name) may not be unique. If this assumption is not met by a particular COBOL program, then it is still possible to produce correct HIBOL, but the renaming process will be made more difficult and the HIBOL that is produced may be redundant in some respects. It might be better in this case to simply drop all but one of the definitions for the data-sets that correspond to data fields that do not contain different information. However, recognizing when two data fields are redundant would be quite difficult.

6.1.3 The COUNT Operation

The COUNT reduction operator is a source of difficulty for the current implementation. It is unlike any of the other reduction operators in that it does not require any data values as input. This is reflected in the analyzed plan by the fact that an augmentation that calculates a count will have a DUMMY temporal input (see, for example, Figure 30). All that controls the operation of the count augmentation is the predicate associated with that DUMMY temporal input. Logically, the COUNT operation in the analyzed plan takes a predicate as an argument and not an object.

The semantics of HIBOL also acts as if the COUNT operator took a predicate as its argument. The syntax of HIBOL, however, demands that the COUNT operator take a data-set as its argument. The COUNT operator works as if it counts the number of times that the predicate:

```
(PRESENT data-set-name)
```

is TRUE.

There are two reasons why the syntax of HIBOL demands that the COUNT operator take a data-set instead of a predicate as its argument. First, it is consistent with the syntax for the other reduction operators. Second, and more importantly, it is very difficult, in general, to count the number of times an arbitrary predicate is TRUE. For example, in order to calculate the number of times the *negation* of a PRESENT predicate for a particular data-set is TRUE, the program would have to subtract the number of data items that actually appear for that data-set from the total number of possible data items for that data-set. It is not obvious how the program could calculate the total number of possible data items for a data-set, in general. Additionally, it is fairly easy to produce predicates that are even more difficult to handle than the negation of a PRESENT predicate.

Since a COBOL program can count up arbitrary things, it will not be possible to produce HIBOL COUNT operators in a reasonable fashion for all possible counts appearing in COBOL programs. Even in the cases in which the count is expressible in HIBOL, it is difficult to produce a data-set name to use as the argument to the HIBOL COUNT operator. Currently, the symbolic evaluation phase uses the predicate associated with the DUMMY temporal input to the count augmentation as the argument to the COUNT operator. The HIBOL production phase then attempts to convert that predicate into a PRESENT predicate, and then use the data-set that is the argument to the PRESENT predicate as the argument to the COUNT operation. For example, in PAYROLL, the predicate that is associated with the DUMMY temporal input to the count augmentation is

```
(NOT (EOFP (SEFO HOURLY-WAGE-IN)))
```

This predicate easily converts to

```
(PRESENT HOURLY-WAGE)
```

using the techniques discussed in the previous chapter, and eventually produces

```
(COUNT OF HOURLY-WAGE)
```

as the final form of the COUNT operator in HIBOL syntax.

This technique, however, is not very robust. In some instances, the predicate produced may contain the conjunction or disjunction of several PRESENT predicates. In other cases the predicate may contain a predicate that cannot be reduced to any PRESENT form. There are two other processes that might be used instead of the one discussed above to determine the data-set that should be used as the argument to a COUNT operator, although neither of them are very appealing.

First, it is possible to make an assertion for the in-case of each segment of the analyzed plan which indicates its control environment.[2] Then, when an augmentation is located during the symbolic evaluation which performs a count operation, the predicate that controls the count could be compared to the control environment of all the other augmentations with the hopes that it will find one with the same truth value. Then, if that augmentation has any output that already corresponds to a data-set, then that data-set could be used as the argument to the HIBOL COUNT operator.

This technique has two shortcomings. First, it is difficult in general to tell when two predicates have the same truth value unless their surface syntax happens to be identical. This is made easier by the fact that the simplification techniques that are used tend to canonicalize the predicate expressions, but this alone is not sufficient to insure that predicates with the same truth value will be recognized as such. Second, even if a control environment is found that does match the controlling predicate of the count, the data-set that is produced by that augmentation might have no conceptual connection with the count operation whatsoever. The use of that data-set name as the argument to the COUNT operation might, therefore, be highly non-mnemonic, although at least it will be a data-set name that already appears in the HIBOL program.

A second possible technique is to simply build a dummy data-set, defined in the VARIABLE SECTION of the HIBOL program, that can be used as the argument to the COUNT operator. The expression for this dummy data-set that would appear in the COMPUTATION DIVISION as a conditional with a single clause in which the predicate is exactly the one that controlled the count augmentation in the analyzed plan, and the object is just some dummy constant.

This technique has the advantage that it will work whenever it is possible to produce a HIBOL COUNT expression at all because it is always possible in those circumstances to produce the needed conditional expression in HIBOL. It has the disadvantage that it introduces a data-set name that is alien to the original program for which the SATCH system user will not be able to supply a mnemonic name because it has no real world analogue. Also, the conditional expression for this newly introduced data-set contains an arbitrary constant that also has no real world significance. Alternately, the conditional expression could be used directly as the argument to the COUNT operator. This eliminates the necessity for the extra data-set name, but does not eliminate the arbitrary constant. Also, the resultant data-set definition for the COUNT would appear needlessly complex. Either way, the HIBOL code produced using this technique may look rather stilted to a

[2] This is a possible change to the current system that has certain advantages independent of the problem with COUNT operations.

human reader.

In summary, there is no single technique for producing COUNT expressions that is satisfactory in all cases. Perhaps the best approach to this problem is to use the three techniques described above in order, first trying the technique that is most specific but which produces the most mnemonic HIBOL code, and going to increasingly general techniques that produce less and less mnemonic code as the more specific techniques fail. In this way, the best possible code will always be produced, although the average cost of producing HIBOL code for COUNT operators will be substantially increased.

6.1.4 Subexpression Aliasing

It is often desirable to define output data-sets in terms of other output data-sets. This can simplify the definition and increase its readability. For example, the definition of TOTAL-GROSS-PAY in the HIBOL program for the PAYROLL example (see Section 1.4), without the use of other output data-sets is:

```
TOTAL-GROSS-PAY IS SUM OF (HOURLY-WAGE * 40.)
```

Through the use of output data-sets in this definition, it can be simplified to:

```
TOTAL-GROSS-PAY IS SUM OF GROSS-PAY
```

The second expression is both simpler and more mnemonic. Both expression are totally valid HIBOL expressions for the same computation. The difference is strictly one of style.

Unfortunately, it is difficult to use output data-sets in the definition of other output data-sets. The difficulty lies in the recognition of those cases where it is possible and/or desirable to do so. Several techniques have been tried to date, none of which was found acceptable. After a few of these have been discussed, a new but untried solution will be presented.

One possible solution to this problem is to use the output data field names corresponding to the desired output data-set names in the assertions formed in the symbolic evaluation of the analyzed plan whenever possible. The analyzed plan for a program contains information that indicates at which points in the program assignment of values to data fields takes place. Therefore, every time an assertion is made, it is possible to replace any subexpression of that assertion with a data field name if that data field has been assigned the value of that subexpression. Then, in the HIBOL production phase, it is simple to form definitions for output data-sets in terms of other output data-sets because the assertions for data fields will already be expressed in terms of other output data fields.

However, there are two problems with this technique. First, there is the trivial problem that many data-sets will end up being defined as themselves. For example, the HIBOL expression for GROSS-PAY in the example above will be

```
GROSS-PAY IS GROSS-PAY
```

This can be eliminated by a special check in the symbolic evaluation phase to see that this does not occur, but the check is messy and not very elegant. A second and more difficult problem is that there is no guarantee that after a data field is used to define another data field it is not assigned a different value before it is written. If such redefinition does occur, then one data-set will end up defined in terms of some data-set name that no longer corresponds to the same subexpression that it replaced. Elimination of this problem would be quite difficult.

Another possible technique is to keep a global association list between subexpressions and data field names. This list would be compiled during the symbolic evaluation of the analyzed plan. Each time an assignment point is reached in the analyzed plan, an entry is made in the table. Then, in the HIBOL production phase, the expressions are scanned for any subexpressions for which there is an entry in the association list, and if one is found, it is replaced with the corresponding data field name.

This technique makes it easy to eliminate the problems cited for the other technique above, but it introduces new problems of its own. First, a subexpression that could have been replaced with a data field name while in the symbolic evaluation phase may not still be in its original form by the time the expression makes it to the HIBOL production phase, because it has been modified by one of the simplification transforms discussed in Chapter 5. Therefore, although it would have been desirable to replace a given subexpression, it no longer appears verbatim and can no longer be found. Second, it is now possible to find subexpressions that *do* match expressions in the association list that it is *not* desirable to replace with the corresponding data field name because the expression in which it is found conceptually has nothing to do with that data field name, and the resultant code would not be mnemonic at all.

A third technique that has not as yet been tried is to simply check all data-set definitions against one another just before the final HIBOL expressions are written into the COMPUTATION DIVISION looking for matching expressions. As compared to the technique described above, this technique reduces the chance that a subexpression that should be replaced by a data-set name will be missed, but still has the problem that certain subexpressions may be replaced by data-set names to which they do not conceptually correspond. A second problem is that the search for matching expression is quite expensive.

In summary, although it would be nice to be able to produce HIBOL in which some output data-sets are defined in terms of other output data-sets, until and unless some technique for doing so is developed that is better than any of the techniques discussed above it is probably not worth the trouble. The current implementation of the translation process expresses all output data-set definitions in terms of input data-sets only.

6.2 Possible Extensions

This section contains a discussion of two possible extensions to the current domain of applicability of the translation process; indexed file access and formatted output reports.

6.2.1 Indexed File Access

One construct that is often used in COBOL programs that cannot currently be translated into HIBOL is the use of indexed data files. Indexed files can be accessed in either sequential or random order. Both of these usages can be translated into HIBOL fairly easily as long as the COBOL program in which they appear still falls into one of the three basic categories of programs that the translation process is currently designed to handle.

The most significant difference between the translation of COBOL programs that access an indexed file and those that don't is that the predicates that are produced in the symbolic evaluation of the analyzed plan will contain INVALID-KEYP predicates as subexpressions when the indexed file is accessed randomly. Recall that the INVALID-KEYP predicate is TRUE if the record associated with the NOMINAL KEY requested by the random read does not appear in the file. The INVALID-KEYP concept in COBOL very closely corresponds to the HIBOL concept of a data value not appearing in the actual key space for a particular data-set. Therefore, the INVALID-KEYP predicates are handled by simply replacing them with the negation of FILE-PRESENT predicates in the HIBOL production phase as is currently done with EOFP predicates, and the remainder of the translation process continues as always. The accessing of indexed files in sequential order should require only the most trivial changes (if any) to the translation process.

Translation into HIBOL of COBOL programs that include the random access of an indexed file that does not contribute to the main read loop of the program is made trivial by the change given above. This construct will most often arise in programs that access library files that contain certain additional pieces of information. For example, a program that processes payroll, in addition to calculating GROSS-PAY, may need to access the employee name that corresponds to a given employee-number. The employee names might be kept in a library file indexed by employee-number. The INVALID-KEYP predicates that result from the accessing of the library file would be handled as described above, and the translation of such a program should proceed smoothly. This is an example of a construct that can be added to a COBOL program without changing the basic structure of the program and therefore could be incorporated into programs of any of the three basic types without affecting the basic category into which the program falls.

There are two additional COBOL program scenarios that involve indexed files which do appear in the main read loop of the program (and therefore do affect the basic structure of the program) that could be translated into HIBOL if the simple change described above were incorporated into the translation process. The first of these involves the random accessing of an indexed file combined with sequential access of a normal sequential file. The second involves the random accessing of an indexed file combined with sequential access of an indexed file. Both of these constructs can only be incorporated into programs which perform intersections and have the effect that a program into which one of them is incorporated can now be viewed as a program which uses only a single data file to drive the computation instead of two (or more) as is usually the case in a program which performs an intersection. Therefore, two programs which perform the same computation, one of which uses only sequential files and the other of which falls into one of the above scenarios, have a different basic structure and do not fall into the same basic program category.

In the first possible scenario, two main data base files contribute data field values to the same computation as discussed in the previous chapter except that one of the files is an indexed file that is randomly accessed. In this scenario, the program loops through the sequential file. For each record in that file, it performs a random read on the indexed file using the key field values obtained from the record read in the sequential file as the NOMINAL KEY for the random access read. For example, consider the COBOL code fragment for a modified version of PAYROLL shown in Figure 46. In this example, HOURS-WORKED-IN is a sequential file, while HOURLY-WAGE-IN is an indexed file that is randomly accessed. Note that the figure does not contain the components of the DATA DIVISION that are required to specify that HOURLY-WAGE-IN is an indexed file with hourly-wage-key acting as the NOMINAL KEY.

Fig. 46. COBOL Fragment with One Sequential and One Indexed File

```
mainline SECTION.
        READ hours-worked-in AT END GO TO end-of-job.
        MOVE employee-number OF hours-worked-rec TO hourly-wage-key.
        READ hourly-wage-in INVALID KEY GO TO mainline.
        MULTIPLY hourly-wage BY hours-worked GIVING gross-pay.
        MOVE employee-number OF hours-worked-rec
          TO employee-number OF gross-pay-rec.
        WRITE gross-pay-rec.
        GO TO mainline.
end-of-job SECTION.
```

Fig. 47. COBOL Fragment with Two Indexed Files

```
mainline SECTION.
        READ hours-worked-in NEXT RECORD AT END GO TO end-of-job.
        MOVE employee-number OF hours-worked-rec TO hourly-wage-key.
        READ hourly-wage-in INVALID KEY GO TO mainline.
        MULTIPLY hourly-wage BY hours-worked GIVING gross-pay.
        MOVE employee-number OF hours-worked-rec
          TO employee-number OF gross-pay-rec.
        WRITE gross-pay-rec.
        GO TO mainline.
end-of-job SECTION.
```

The second possible scenario is almost identical to the first scenario except that both files are indexed files, although one of them is read sequentially. COBOL provides for the sequential access of indexed files through the use of the NEXT RECORD clause in the READ statement (see [22]). The other file is read in random order, using the key field values from the record read in the first file as the NOMINAL KEY for the random read. For example, see the COBOL code fragment shown in Figure 47. Again note that the figure does not contain the components of the DATA DIVISION that are required to specify that HOURLY-WAGE-IN is an indexed file with hourly-wage-key acting as the NOMINAL KEY and that HOURS-WORKED-IN is an indexed file that will be accessed sequentially.

It is important to note that in both of these two scenarios, although two input files are contributing data values to the same computation, the two files need not be sorted in the same order. These are probably the two cases in which the sorting constraint mentioned in the previous chapter can be most easily eliminated.

In summary, the important point that makes possible the translation of COBOL programs that incorporate indexed access reads that are randomly accessed is that the INVALID-KEYP predicates are replaced with the negation of FILE-PRESENT predicates. As long as the NOMINAL KEYs that are used to access an indexed file are generated in a fashion that allow the program to be classified as one of the three allowable types, and all of the other assumptions about the COBOL program are met, the inclusion of indexed files in a COBOL program should pose no significant problems to the translation process.

An interesting by-product of the use of an indexed file in a COBOL program is that the COBOL programmer must specify the key fields for that file in the DATA DIVISION. The translation process can make use of this information to avoid the necessity of asking the SATCH system user for the key fields or sort order for that file, reducing the length of the data division query subprocess.

6.2.2 Formatted Output Reports

COBOL programs that produce formatted output reports differ from the COBOL programs considered so far in two important ways. First, it is most often the case that the production of the formatted report will call for CWRITE operations in several different places in the program (or analyzed plan for the program) all acting on the same file-object, while a data file is usually produced with one or at most a few different CWRITE operations in the program (all of which are executed in mutually exclusive control environments). Second, in addition to the usual computation to derive the values of the data fields in the formatted report, there will also be computation used solely to control spacing, page ejects, choice of literal strings, etc.

Because of these differences, it will no longer be sufficient to symbolically evaluate the analyzed plan and then simply pass on the assertions for data flows used as arguments to CWRITE operations to the HIBOL production phase. The symbolic evaluation can proceed as always, but a second pass over the analyzed plan will have to be made in which the pattern of CWRITEs performed on a given file-object is examined. The different portions of the pattern of CWRITEs that are found will contribute to different components of the HIBOL formatted report feature.

In HIBOL, a formatted report is broken down into several components, such as report headings, report footings, page headings, page footings, typical lines, etc. (See [30] or [18] for a discussion of the HIBOL document facility.) A typical pattern of CWRITEs for a formatted report might be broken into these components as follows. Report headings and footings would appear as a series of CWRITEs that occur outside of the main loop of the program, with headings coming before and footings after the temporal composition that represents that loop. The main CREAD loop that drives the entire computation may appear nested within a second loop that counts up to fifty (or some similar constant) in order to control page ejects. Page headings and footings would appear as a series of CWRITEs within the temporal composition that represents the page eject loop, but not within the nested CREAD temporal composition. The CWRITE that produces the typical line would then appear within the nested loop.

The second pass over the analyzed plan would have to keep track of its current location in the analyzed plan relative to the main temporal compositions. Then, when a CWRITE is located, this information would be used to determine which component of the report the output of the CWRITE should be relegated to. The assertions about the input data ports to the CWRITEs, formed during the first pass, are used as always to determine the nature of the data values output by each CWRITE. After the second pass is complete, the overall pattern of the CWRITEs, and therefore of the report, can be determined.

This entire process should be simplified by the fact that there probably are not very many different overall patterns that need to be recognized; perhaps at most a few dozen or so. The exact number needed is not now known, but can be empirically determined through further research.

The translation of COBOL programs that produce formatted reports also calls for a simplifier for expressions that contain combinations of SUBSTRING and CONCATENATE operations acting on STRING objects. These expressions will arise in the program for the control of spacing and choice of literal strings. The simplifier would reduce such expressions into literal constants whenever possible. Such a simplifier should not be difficult to produce.

As the final step in the translation of COBOL programs that produce formatted reports, the HIBOL syntax for the DOCUMENT SECTION of the DATA DIVISION would have to be produced. This syntax is somewhat elaborate, but should not be overly difficult to produce once the pattern of the reports is known and the expressions for the string operations have been simplified.

In summary, although the translation of COBOL programs that produce formatted output reports into HIBOL requires more elaborate processing of the analyzed plan, additional simplification techniques, and a more elaborate HIBOL production phase, it is not beyond the reaches of current technology. None of the new features of the translation process described above should be overly difficult to produce. Thus, this increase in the domain of applicability of COBOL to HIBOL translation could be achieved through a moderate engineering effort.

7. Critique of the SATCH System

In the previous chapter, several features of the current implementation of the translation process were discussed, and suggestions made as to how the translation process could be modified to improve its performance. In this chapter, the current implementation of the entire SATCH system is discussed, with some suggestions as to how the system performance could be improved by making changes at this more global level.

7.1 Semiautomatic versus Automatic Translation

Although the word "semiautomatic" appears in the title of this thesis, the current implementation of the SATCH system essentially performs the automatic translation of COBOL programs into HIBOL. Of the three major components in the system, the COBOL parser, the plan analyzer, and the translation process, only the translation process utilizes any human input.

The translation process utilizes human input in two places. First, the key fields for the files manipulated by the COBOL programs must be specified. Although the SATCH user is currently asked to supply this information for every COBOL program that is translated, the key fields for the data files remain constant throughout an entire data processing system. Therefore, the SATCH system could be changed so that the key field information for a data processing system is input only once, and then used in the translation of all the COBOL programs in that system. This would significantly reduce the amount of human input required by the system.

The second form of human input is utilized in the HIBOL production phase of the translation process to specify which data field(s) in a file should be used to replace FILE-PRESENT predicates with PRESENT predicates. This information, however, is only needed to increase the readability of the resultant HIBOL program, and is not required to insure the correct semantics of the HIBOL program. Therefore, it would be possible to eliminate this input without affecting the correctness of the translation.

Therefore, the human input required by the system to translate the current domain of COBOL programs is minimal. The expansion of the domain, however, might call for an increase in the amount of human intervention as discussed below.

7.2 Using Analyzed Plans

Given that the task at hand is to translate a process described in one language (COBOL), into the same process described in a more abstract language (HIBOL), the abstraction process is of the utmost importance. Currently, most of the abstraction is performed by the component of the SATCH system that produces the analyzed plan. This component uses general methods to abstract away the details of implementation in the source language (in this case, COBOL). The component of the system that translates the analyzed plans into HIBOL does a certain amount of abstraction, however it uses special case techniques that are specifically designed around the features of the target language (HIBOL). The breakdown of the abstraction process into these two components raises a key question; Are the general method abstractions that are made in the analysis of plans useful for the translation of COBOL programs into HIBOL, or would it be better to apply special case abstraction techniques right from the beginning?

The answer is that the abstractions contained in an analyzed plan are exactly those that are needed for the translation of COBOL programs into HIBOL. In general, programs can be abstracted in several different ways producing program representations containing very different types of objects and operations on those objects. For example, a program can be broken down into subprograms that each perform a specific task as is done in FORTRAN [19] and PL/1 [33]. Or it can be broken down in terms of increasingly abstract data objects and operations defined to operate on those objects as is done in Alphard [35], SMALLTALK [13], and CLU [15]. Or it can be broken down into independently acting agents that wait to be activated depending on the current environment as is done in CONNIVER [29] and PLASMA[10]. Finally, it can be broken down into data flows and operators that act on values carried by those flows as is done in VAL [1] and HIBOL. Analyzed plans also express programs using this last paradigm. Therefore, a program expressed as an analyzed plan is broken down into the same abstract components as a program expressed in HIBOL. This does not mean that any program that can be expressed as an analyzed plan can be expressed in HIBOL, but it does mean that for those programs that can be expressed in HIBOL the analyzed plan representation of that program more closely corresponds to the HIBOL representation than could any representation which is based on one of the other abstraction techniques. This makes the abstraction of a COBOL program into an analyzed plan a very provocative first step in the translation of that program into HIBOL.

In spite of the fact that an analyzed plan is ideally suited to the translation of COBOL programs into HIBOL for the reason stated above, the use of analyzed plans in this process has certain drawbacks. First, an analyzed plan is an unwieldy representation of a program from the standpoint of human interaction. It was designed to make it easier for a computer program to understand another program, not to make it easier for a person to understand that program. Therefore, should it become necessary or desirable to involve a human in the portion of the translation process that involves the analyzed plan, the plan itself would be a particularly poor choice for the vehicle of discourse between the human and the program. Either the analyzed plan would have to be temporarily translated into some form that the human can interact with, or the possibility of human interaction in that portion of the translation process would most likely have to be abandoned. Of lesser importance, the fact that the analyzed plan representation is unwieldy increases the difficulty of debugging the portion of the translation process that interacts with it.

A second shortcoming of the use of analyzed plans in the translation of COBOL programs into HIBOL is that the program that produces the analyzed plan from the surface plan does not currently incorporate enough knowledge about the interaction of input/output operations with the data flows that represent file-objects. The original test bed for analyzed plans was the FORTRAN Scientific Subroutine Package. These subroutines, in general, perform numerical analysis operations on matrices and other data objects, but do not perform any input/output operations. Therefore, sufficient knowledge about such operations was never incorporated into the analysis process. This shortcoming has led to the production of analyzed plans, in both the DBINIT and PAYROLL2 examples, containing temporal compositions with a single generating augmentation, which contains essentially all of the computation performed by the program, and a single termination as their only two subsegments. Such analyzed plans are more difficult to work with than ones in which there are several generating and/or consuming augmentations each of which performs a simpler function. The analysis process could be modified to incorporate the necessary knowledge with a (hopefully) moderate amount of effort.

Notwithstanding the shortcomings of analyzed plans cited above, the translation of COBOL programs into HIBOL would be much more difficult, if not impossible, without the use of them or some similar program representation. The current implementation of the translation process relies upon them implicitly and could not be reasonably modified to work should they be abandoned.

7.3 Future Direction for the Translation Process

The current implementation of the translation process was designed with the subset of COBOL programs that it currently can translate in mind. Expansion of the translation process to operate on a larger domain of COBOL programs, except in those cases cited in the previous chapter, might be very difficult. The purpose of this thesis was to show the *feasibility* of the translation of COBOL into HIBOL, not to present a final solution to the problem. The next attempt to build a COBOL to HIBOL translator should replace the current implementation of the translation process with one that incorporates the work currently being done by Rich and Brotsky at MIT. The remainder of this section describes how that implementation of the translation process might work.

Charles Rich, in his PhD thesis [26], proposed a method of further abstracting analyzed plans by recognizing standard program cliches within them. He calls such abstraction "plan recognition by inspection". The process proceeds as follows. First, the analyzed plan is converted into another representation called the "plan calculus". This process is relatively simple. The plan calculus is a way of expressing a program in a hierarchical structure identical to analyzed plans except that the primitive elements in the calculus are essentially propositions in first order predicate calculus. The plan is converted into this representation to aid in the recognition of the plan cliches and facilitate logical reasoning about the plan.

After the plan is translated into the plan calculus, the recognition process attempts to match structures in the plan for the program with precompiled patterns taken from a plan library. The plan library contains cliches for both common computational abstractions and common data abstractions. A key feature of the matching process is that a given fragment of the plan can simultaneously be used to fill roles in several different library cliches. This allows the program to be examined from multiple viewpoints. A key feature of the plan library itself is that the plan cliches are built into a taxonomy so that certain cliches can be viewed as extensions of other cliches or as specializations of more general cliches with added specifications. Another key feature of the plan library is that there are names associated with all of the computation and data abstractions. Therefore, a system using this plan recognition scheme can converse with a human using the same vocabulary that is used in everyday conversations among expert programmers. Finally, it is intended that the plan cliches that appear in the library could be used equally easily for program analysis or program synthesis.

As part of his PhD research, Rich designed the plan calculus and the taxonomy for library cliches, and constructed a sample library containing a couple hundred entries. Currently, a joint effort is underway by Rich and Brotsky to implement a program to convert analyzed plans into the plan calculus. They are also putting the plan library into an on-line data base. Brotsky, as part of his Master's research, intends to design and implement a program that will automatically recognize instances of library cliches in a program represented in the plan calculus.

When the programs described above are implemented, the translation process of analyzed plans into HIBOL can be rewritten to take advantage of them. In the simplest view of this scheme, all that would be required is that the plan library be expanded to include the cliches that appear in COBOL programs which closely correspond to HIBOL constructs. Then, once the cliches are recognized, it would be a fairly trivial process to convert them into HIBOL syntax.

This scheme has several advantages over the current technique used in the translation process. First, instead of having all the special case knowledge needed for the translation embedded within LISP code, that knowledge would be contained within the plan library. This makes the knowledge much more accessible, and far easier to extend and modify. Second, it is hoped that this scheme could capture more pertinent knowledge and therefore provide for the translation of a much larger scope of COBOL programs. This was foreshadowed in the earlier discussion of a possible extension to the current translation process that would allow programs that produce formatted output reports to be translated. The second pass over the analyzed plan that was described in that discussion can be viewed as an intermediate point between the current technique and the one described in this section. A third advantage of this scheme is that the knowledge gained during its implementation could be applied to the translation of other procedural languages into other higher level languages, with the implementation of these translation systems requiring a relatively minor amount of additional effort. For example, it might be possible to apply such techniques to the translation of a certain subset of FORTRAN programs into APL [24].

Unfortunately, it is doubtful that it will be possible to translate COBOL programs into HIBOL using just the simple scheme outlined above. First, it is doubtful that the component of the system that is responsible for the recognition of plan cliches in the plan calculus could successfully operate on a plan calculus representation produced from an analyzed plan in which almost all of the computation of the program is contained within a single augmentation. Such analyzed plans were mentioned in the previous section. The analysis process would have to be extended to produce better analyzed plans for programs that perform input/output operations on file-objects before this scheme would be possible. As stated before, this extension of the analysis process should not be overly difficult.

Even if the analysis process were so extended, it is doubtful that the recognition process would ever recognize all of a plan in terms of known cliches for anything other than the most trivial programs. Therefore, this system would probably call for human assistance for part of the recognition process. Unfortunately, the plan calculus is no better suited as a vehicle of discourse with humans than the analyzed plan representation. An interface would have to be built to intercede between the recognition process and the human user. The construction of such an interface is made easier by the fact that the cliches in the library have names associated with them that can be used in the man/machine dialogue. A program synthesis system currently being worked on by Rich and Waters requires a similar interface. The interface routine developed as part of that project could conceivably be modified and transported to the COBOL to HIBOL translation process.

In summary, the possibility of applying the method of plan recognition by inspection to the translation of COBOL programs to HIBOL is a provocative one. Although it poses some problems that need to be overcome, it offers promise for the production of a system with much greater performance than the current system. Such a system might well incorporate both a symbolic evaluation of the analyzed plan, similar to the one used in the current system, as well as the recognition of plan cliches.

7.4 Translation of HIBOL into COBOL

Although the predominant motivation for this thesis is to show the feasibility of translating COBOL programs into HIBOL, it is necessary to mention the possibility of the translation of HIBOL programs back into COBOL in order to impart an overall perspective.

PROTOSYSTEM I [27] is an automatic programming system, developed by the Automatic Programming Group at MIT, which can translate HIBOL programs into compilable PL/1 code and the corresponding IBM JCL needed to run the resultant programs. In general, the only assumptions made by the system about the target language is that it is some high level procedural language that supports input/output operations to sequential and indexed data files. The exception is the final component of the system which produces the actual PL/1 syntax for the computation. All that is required to allow the system to produce COBOL programs is to replace the PL/1 syntax generator with a COBOL syntax generator. A syntax generator for COBOL should not be overly difficult to produce.

As stated in Chapter 1, there are some problems with the unconstrained use of PROTOSYSTEM I to produce COBOL programs from HIBOL. To understand the problems and how they can be circumvented, a slightly more detailed view of PROTOSYSTEM I is needed.

A primary design goal of PROTOSYSTEM I was the ability to produce highly efficient code from a HIBOL program. To accomplish this end, PROTOSYSTEM I is broken into two major components. The first of these, the "design optimizer", is responsible for determining the desired "data aggregation" and "computation aggregation" for the application. The data aggregation specifies which data-sets should be grouped together in the same file, and what type of files there should be (indexed or sequential). The computation aggregation specifies which operations on the data files should be grouped together in the same program. The second major component of the system, the "code optimizer", uses the output of the design optimizer and determines the desired implementation of the programs themselves.

The design optimizer represents the portion of the system that does most of what is usually called automatic programming. The code optimizer performs a more well understood function; one strongly resembling that of an optimizing compiler. Upon completion of the PROTOSYSTEM I project, there remained certain research issues with respect to the design optimizer that were not completely resolved. The code optimizer that was developed produced PL/1 code with very good run-time characteristics.

Within the context of the use of PROTOSYSTEM I as a component of the SATCH system, it is highly desirable to produce a single COBOL program for a single HIBOL program, and the COBOL program should operate on the same data files that were used by the original COBOL program from which the HIBOL was produced. Therefore, the design optimizer component of PROTOSYSTEM I is not required, since the data and computation aggregation used by the code optimizer should be exactly those specified in the original COBOL program. This constrained use of PROTOSYSTEM I, within the context of the SATCH system, should result in output COBOL programs that are highly run-time efficient.

Appendix I - Plan Primitives for COBOL Programs

In this appendix, all of the primitive functions that can appear in a plan that is produced from a COBOL program are explained along with the number and type of their arguments. Most of these primitive functions perform standard operations that commonly appear in any programming formalism. These standard functions are included here for completeness. The remainder of the functions perform operations that are much less standard. Particular attention will be given to the latter.

Each of these functions can be viewed as a black box, with a number of explicit inputs and outputs where the outputs are related to the inputs via the function given. They should not be thought to *return* a value in the usual LISP sense, but rather to *produce* a value that is carried from the function via explicit data flow. Also, some of the functions may produce more than one value as a result of their operation. In addition, certain of the functions that perform operations on file objects cause side effects. This will be discussed in greater detail below. For these reasons, these primitives are not actually functions in the usual sense.

I.1. Boolean Primitives

Each of the following functions result in the production of a single bit boolean. The input arguments are of various types.

AND: Binary operator that performs the standard logical AND. Both arguments are booleans.

EOFP: Takes a file object as input and produces TRUE if the next CREAD of the file will produce an end of file condition, and FALSE otherwise. The file object itself is unaltered by this test.

EQUAL: Binary operator that performs the standard EQUAL function. The input arguments can be either both numbers or both strings. When the arguments are strings, a standard collating sequence is used.

GREATERP: Binary operator that performs the standard GREATER-THAN function. The input arguments can be either both numbers or both strings. When the arguments are strings, a standard collating sequence is used.

INVALID-KEYP: Takes a file object as input and produces TRUE if the next CREAD of the file will produce an invalid key condition, and FALSE otherwise. The file object itself is unaltered by this test. This is used with indexed files only.

LESSP: Binary operator that performs the standard LESS-THAN function. The input arguments can be either both numbers or both strings. When the arguments are strings, a standard collating sequence is used.

NOT: Unary operator that performs the standard logical NOT. The input argument is a boolean.

OR: Binary operator that performs the standard logical OR. Both arguments are booleans.

I.2. Arithmetic Primitives

All arguments to and results from the following functions are numbers.

DIFFERENCE: Binary operator that produces the result of subtracting the second argument from the first.

MINUS: Standard unary minus operation.

PLUS: Binary operator that produces the sum of two numbers.

REMAINDER: Binary operator that produces the remainder after dividing the first argument by the second argument an integer number of times. In other words, it produces the first argument MODULO the second argument.

TIMES: Binary operator that produces the result of multiplying the two arguments.

I.3. String Operators

Each of the following functions result in the production of a string. In this system, a string is a special object type formed by an invocation of STRING. The input arguments are of varying types.

CONCATENATE: Binary operator that produces a string formed by immediately following the value of the first argument with the value of the second argument. Both arguments are strings.

STRING: Unary operator that forms a string object from a sequence of characters.

SUBSTRING: Takes three arguments and produces a string. The first argument is the string from which the substring will be taken. The second and third arguments specify the first and last characters of the first argument to be included in the resultant substring, respectively.

I.4. File Operators

All of the following functions take a file object as their first argument. The file object should be looked upon as a pointer into a file of data records. The pointer contains information about the next record to be accessed (if any) as well as certain status information about the file. Some of the following functions update the file object as a result of their operation. This is done by merely having an output data flow produced that is different than the incoming data flow for the file object. This is analogous to the way in which all other data values are handled within a plan.

In addition, however, the file that is pointed to by the file object may be side effected by the operation of the function. For example, the CREWRITE function will destroy information stored in a particular record of a file and replace it with new information. The file is permanently altered by this operation, and the old version of the file is no longer available. These are the only functions produced from a COBOL program that can cause side effects. It is clearly stated in the following function descriptions exactly which functions cause side effects.

CCLOSE: Takes a file object as its only argument and produces an updated file object. In addition, the file is side effected such that it can no longer be accessed via any file operator except one of the following OPEN functions.

COPENI: Takes a file object as its only argument and produces an updated file object. In addition, the file is side effected such that it can now be accessed by CREAD. That is, the file is opened for input only. The file object is set to point to just before the position of the first record.

COPENIO: Takes a file object as its only argument and produces an updated file object. In addition, the file is side effected such that it can now be accessed by CREAD and CREWRITE. That is, the file is opened for input/output access. The file object is set to point to just before the first record.

COPENO: Takes a file object as its only argument and produces an updated file object. In addition, the file is side effected such that it can now be accessed by CWRITE. That is, the file is opened for output only. The file object is set to point to just before the position of the first record.

CREAD: Takes a file object as its only argument and produces an updated file object as well as an arbitrary number of data values taken from the record in the file that is specified by the file object. The record that the data values are taken from depends upon several factors. If the file specifications given in the original COBOL program specify sequential access for the file, then each CREAD will access the record that is currently pointed to by the file object and then update the file object to point to the next contiguous record. Since the COPEN causes the file object to point just before the first record in the file, the first CREAD will cause the first record in the file to be accessed. If the file specifications given in the original COBOL program specify random access for the file, then each CREAD will attempt to access the record in the file that corresponds to a particular set of values of the key fields. The set of values of the key fields that will be used to specify the record to be accessed is contained within the file object. If the particular key set specified does not correspond to any record that actually exists within the file, then INVALID-KEYP will produce TRUE, and the CREAD will not take place (assuming the original COBOL text represents a valid COBOL program). If a random access CREAD is successfully completed, then the file object produced will point to the record in the file that was just accessed. This ensures that a subsequent CREWRITE will access the correct record.

CREWRITE: Takes a file object and an arbitrary number of data values as arguments and produces an updated file object. In addition, the file is side effected by overlaying the record in the file specified by the file object with the argument data values. This is used with indexed files opened via COPENIO only.

CWRITE: Takes a file object and an arbitrary number of data values as arguments and produces an updated file object. In addition, the file is side effected by placing a record in the file at the place pointed to by the file object. The record is composed from the argument data values. This is used with files opened via COPENO only.

NTERPRI: Takes a file object and an integer as arguments and produces an updated file object. In addition, the file is side effected by placing the integer number of end-of-record marks in the file at the place pointed to by the file object. For normal data files the second argument is always 1 and NTERPRI is invoked once before each invocation of CWRITE. The use of CREWRITE does not require the use of NTERPRI because the end-of-record marks should already appear in the file.

Appendix II - How to Run SATCH

This appendix contains the instructions for running the SATCH system. Included are the file names and locations of all pertinent programs, the naming conventions of the data files used, and a very brief description of some of the more important top-level program functions. All programs are now on ML. Although all the programs are currently available and (hopefully) running, there is no guarantee that things will remain in their current state.

To run the COBOL parser, type ":satch;cobpar<cr>". The only pertinent top-level function is RUN. It takes a single argument which is the name of the file which contains the COBOL program to parse. It produces two output files. The first of these, given a second file name of PROG, contains the lisp-like representation of the PROCEDURE DIVISION. The second file, given a second file name of DATA, contains the DATA DIVISION information. For example, the command "(RUN '((DSK DIREC) EXAMPL COBOL))" will parse the COBOL program in "dsk:direc;exampl cobol" and produce the output files "dsk:direc;exampl prog" and "dsk:direc;exampl data". For further documentation for the COBOL parser and/or the answer to any questions about the COBOL parser, contact Glenn Burke (GSB@ML).

To produce an analyzed plan for an output file of the COBOL parser, type ":lisp forpas;<cr>". The pertinent top-level function is PROCESS. It takes a single argument which is the name of the file which contains the PROCEDURE DIVISION output of the COBOL parser to be analyzed. It produces a single output file, given a second file name of PLAN, which contains the analyzed plan. For example, the command "(PROCESS '((DSK DIREC) EXAMPL PROG))" will analyzed the program and produce the analyzed plan in "dsk:direc;exampl plan". For further documentation for the analyzer and/or the answer to any questions about the analyzer, contact Dick Waters (DICK@AI).

To produce HIBOL for an analyzed plan and the DATA DIVISION information, start up a lisp using the initialization file on the FAUST directory. This is done by typing ":lisp faust;faust lisp". Then, one of the following two top-level functions must be run to load the rest of the desired environment: SET-UP-FOR-SATCH-I or SET-UP-FOR-SATCH-C which load the needed LISP source files or LISP FASL files, respectively (SET-UP-FOR-SATCH-C is strongly recommended).

Once the environment is loaded, the most important top-level function is LOAD-TRANSLATE. It takes a single argument which is the first file name of the program to be translated. An attempt will then be made to load the necessary files from the SATCH directory. For example, the command "(LOAD-TRANSLATE 'EXAMPL)" will attempt to load "dsk:satch;exampl plan" and "dsk:satch;exampl data". It is possible to load the files from another device and/or directory by first setting the global variables GLOBAL-DEVICE and/or GLOBAL-DIRECTORY to appropriate values.

LOAD-TRANSLATE will produce two output files, given second file names of FAST and HIBOL, which contain the information passed from the symbolic evaluation phase to the HIBOL production phase, and the completed HIBOL program, respectively. The same functionality can be gained by calling LOAD-DB and TRANSLATE in succession. LOAD-DB takes the same argument as LOAD-TRANSLATE. TRANSLATE takes no argument.

Once LOAD-TRANSLATE has been run on a particular example program, further testing of new versions of the DATA DIVISION Query and/or the HIBOL Production phase(s) can be accomplished by using the top-level function FAST-LOAD-TRANSLATE. This function takes the same argument as LOAD-TRANSLATE. Its operation only differs in that instead of loading in the analyzed plan and performing the symbolic evaluation, it loads the information needed by the DATA DIVISION Query and HIBOL Production phases directly from the files with the second file names DATA and FAST respectively. The two functions FAST-LOAD-DB and FAST-TRANSLATE have the same relation to LOAD-DB and TRANSLATE as FAST-LOAD-TRANSLATE has to LOAD-TRANSLATE.

The entire translation process runs in three different modes which differ only in the amount of information that is written to the terminal. The three modes are controlled by running three top-level functions called VERBOSE-MODE, NORMAL-MODE, and QUIET-MODE. These functions take no arguments. The default is NORMAL-MODE. QUIET-MODE should only be used for batch jobs. VERBOSE-MODE will print out all sorts of intermediate values for variables, and is only useful for trying to debug very severe problems.

Regardless of which mode the program is running in, the terminal will be used to gather information from the user. In all cases where user input is required, the user will be shown a list of data field names and asked to input a list of the desired fields. The user should input the list using the numbers that correspond to the data fields, and not the names themselves. If it is desirable to select none of the fields, NIL is entered. In all cases, the user is given the opportunity to verify the input before the program finally accepts it.

The only remaining top-level function of possible pertinence is DB-WALK. This function is an interactive command interpreter that affords a way to wander around and print out portions of the analyzed plan in a reasonably simple fashion. The set of commands is too large to be discussed here. the definition of the function can be found in "ml:satch;sutil >".

All the source files for the COBOL parser and the translation process are in the archive file "ml:faust;ar0 satch".

References

1. Ackerman, W. B., and Dennis, J. B., VAL - A Value-Oriented Algorithmic Language: Preliminary Reference Manual, MIT Laboratory for Computer Science TR-218 (June, 1979).

2. MetaCOBOL User Guide, Manual No. SM2G-00-10, Applied Data Research (1979).

3. Baron, R. V., Structural Analysis in a Very High Level Language, S.M. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (1977).

4. Barstow, D.. A Knowledge-based System for Automatic Program Construction, *Proceedings of the Fifth International Joint Congference on Artificial Intelligence, Vol. 1* pp. 382-388, M.I.T., Cambridge, Ma., August 1977.

5. Canning, R. G. (ed), Progress Toward Easier Programming, *EDP ANALYZER* 9:1 (September, 1975)

6. Green, C., A Summary of the PSI Program Synthesis System, *Proceedings of the Fifth International Joint Congference on Artificial Intelligence, Vol. 1* pp. 382-388, M.I.T., Cambridge, Ma., August 1977.

7. Hammer, M. M., Howe, W. G., and Wladawsky I., An Overview of a Business Definition System, IBM Thomas J. Watson Research Center Research Report (August, 1973).

8. Hammer. M., and Ruth, G. R., Automating the Software Development Process, in P. Wegner, ed., *Research Directions in Software Technology*, MIT Press (1979).

9. Hartman, J., Restructuring COBOL Programs into Abstract Data Type Modules, University of Texas at Austin Department of Computer Sciences, Software and Data Base Engineering Group, Memo SDBEG-21, (August, 1980).

10. Hewitt, C., How to Use What You Know, *Proceedings of the Fourth International Joint Congference on Artificial Intelligence, Vol. 1*, pp. 189-198, Tbilisi, Georgia, USSR, September 1975.

11. IBM OS Full American National Standard COBOL, Manual No. GC28-6396-5, IBM (1973).

12. Kapur, Deepak, Some Results for Predicate Simplification, MIT Laboratory for Computer Science, Automatic Programming Group Internal Memo, (September, 1976).

13. Kay, A., and Goldberg, A., Personal Dynamic Media, *Computer* IEEE v.10 3:31-41 (1977).

14. Kornfeld, W. A., Ruth, G. R., and Baron, R. V., Proposal for HIBOL Syntax, MIT Laboratory for Computer Science, Automatic Programming Group Internal Memo, (October, 1976).

15. Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheifler, R., and Snyder, A., CLU Reference Manual, MIT Laboratory for Computer Science TR-225 (October, 1979).

16. Long, W. J., A Program Writer, MIT Laboratory for Computer Science, Technical Report TR-187 (1977)

17. Martin, W. A., A Data Set Language and Its Translation into IBM 370 PL/I, MIT Laboratory for Computer Science, Automatic Programming Group Internal Memo, (March, 1972).

18. Martin, W. A., Ruth, G. R., Alter, S., A Very High Level Language for Business Data Processing, *Personal Communication* (1979).

19. McCracken, D. D., *A Guide to FORTRAN Programming*, John Wiley and Sons (1961).

20. Mills, H. D., Software Development, *IEEE Transactions on Software Engineering* SE-2:265-273 (1976).

21. Morgenstern, M. L., Automated Design and Optimization of Management Information System Software, PhD Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (1976).

22. Murach, M., *Standard COBOL (2e)*, Science Research Associates (1975).

23. Pitman, K. M., A FORTRAN to LISP Translator, *Proceedings of the 1979 MACSYMA Users' Conference* pp. 200-214, Washington, DC., June 1979.

24. Polivka, R. P., and Pakin, S., *APL: The Language and its Usage*, Prentice-Hall (1975).

25. Rich, C., and Shrobe, H. E., Initial Report on a LISP Programmer's Apprentice, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report AI-TR-354 (1976).

26. Rich, C., Inspection Methods in Programming, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report AI-TR-604 (1981).

27. Ruth G. R., Protosystem I: An Automatic Programming System Prototype, MIT Laboratory for Computer Science TM-72 (July, 1976).

28. Ruth G. R., Data Driven Loops, MIT Laboratory for Computer Science TR-244 (1980).

29. Sussman, G. J., and McDermott, D. V., From PLANNER to CONNIVER - A Genetic Approach, *Proc. FJCC 41*:1171 (1972).

30. Thomas, G., The Design and Implementation of a Document Facility for Protosystem I, S.B. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (1976).

31. Waters, R. C., Automatic Analysis of the Logical Structure of Programs, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report AI-TR-492 (1978).

32. Waters, R. C., A Method for Analyzing Loop Programs. *IEEE Transactions on Software Engineering* SE-5:237-247 (1979).

33. Weinberg, G. M., *PL/1 Programming Primer*, McGraw-Hill (1966).

34. Weinberg, G. M., Wright, S. E., Kauffman, R., and Goetz, M. A., *High Level COBOL Programming*, Winthrop Publishers (1977).

35. Wulf, W. A., London, R. L., and Shaw, M., An Introduction to the Construction and Verification of Alphard Programs, *IEEE Transactions on Software Engineering* SE-2:253-264 (1976).